Topics for Cucumber JVM:

* Introduction or Why Cucumber.
* Prerequisite.
* What is BDD.
* What is Gherkin.
* Installation.
* Writing Feature File.
* Implementation Steps.
* Hooks.
* Tags.
* Cucumber Options.
* Running Cucumber.
* Page Object.
* New Features.
* Cucumber Assertions.
* Parameters in Cucumber.
* Reporting in Cucumber.
* Dependency Injection.

**Introduction:**

Cucumber is an open-source tool that supports the Behavior Driven Development (BDD) architecture to write acceptance tests for application under test. It enables software teams to collaborate in a way that bridges the gap between business and technical employees, allowing them to do functional validation in a way that is both readable and understandable.

Cucumber-JVM is the official Java implementation. Every Gherkin step is "glued" to a step definition method, which executes the step. The English text of a step is bound using annotations and regular expressions. Cucumber-JVM is compatible with a variety of testing applications. Anything that can be done in Java can be done with Cucumber-JVM. Cucumber-JVM is ideal for black-box, unit functional testing.

**Prerequisite:**

1. BDD and Gherkin
2. JDK
3. Test automation (like JUnit and TestNG)
4. Build process (Maven)
5. IDE (Intellij preferred)

**What is BDD:**

BDD (Behavior-Driven Development) is a set of development approaches targeted at making software development more efficient. It simulates how an application should act from the user's perspective. The main goal of BDD testing is to encourage collaboration among stakeholders like as developers, testers, product managers, and business analysts. Cucumber-JVM is the framework that enables BDD.

BDD guarantees that everyone is kept informed from the start and throughout the process, resulting in better communication overall.

BDD is explained in layman's terms. Everyone has significant insight into the project's development by adopting a basic language that is understood by everybody.

**What is Gherkin and its Syntax:**

Gherkin is a business-friendly test automation and documentation language. Cucumber understands it, and the two work together as a Behavior Driven Development tool.

The fundamental structure of a Cucumber feature file in Gherkin will be demonstrated in this example. The basic syntax of feature files includes various keywords.

The primary keywords are:

* Feature.
* Rule (as of Gherkin 6)
* Scenario.
* Given, When, Then.
* Background.
* Scenario Outline.
* Examples.

**Feature**: this keyword symbolizes that what follows is a basic description or name of the feature being tested or documented.

**Rule:** this keyword is used to express a single business rule that should be followed. It adds to the information about a feature. Several scenarios that relate to this business rule are grouped together by a rule. A rule should include one or more examples that demonstrate the rule in question.

**Scenario**: This term denotes the name or a brief description of the scenario that will be used to test the functionality.

**Given**: this keyword denotes a specific step, or precondition that must be met before proceeding. In the Arrange, Act, Assert paradigm, **Given** represents "Arrange".

**When**: this term is used, it refers to a when step or the behaviour that is being claimed against. In the Arrange, Act, Assert paradigm, **When** represents "Act".

**Then:** this term denotes a then step, or the step in which the outcome of a behavior is validated. In the Arrange, Act, Assert paradigm, **Then** represents "Assert".

To be recognized by the Gherkin parser, all keywords must be on a new line and must be the first word on that line. Following the terms Feature and Scenario, there must be a colon, as seen in the sample below. There is no need for a colon in Given, When, and Then.

**Background:** Occasionally, you'll find yourself repeating the identical Given steps throughout a Feature's situations. Because it's repeated in every scenario, it's safe to assume that those procedures aren't necessary to define the situations; they're just details. By collecting such Given steps in a Background section, you may essentially shift them to the Background.

A Background helps you to give the situations that follow it some context. It can have one or more Given steps, which are executed before to each scenario but after any Before hooks.

**Scenario Outline:** You may use the Scenario Outline keyword to run the same Scenario several times with different values.

**Examples:** A section called Examples must be included in a Scenario Outline. Its stages are viewed as a template that is never executed in its entirety. Instead, for each row in the Examples section under it, the Scenario Outline is executed once (not counting the first header row).

**Installation:**

(Note: Make sure that you have installed Java and Maven - version 3.3.1 or higher)

Step-1: Install the cucumber plugin in Intellij  
  
The IntelliJ IDEA Cucumber plugins make it easy to deal with Gherkin feature files in an IntelliJ project that uses the Cucumber framework. There are plugins available for Java, Scala, and Groovy.

Cucumber for Java IntelliJ plugin includes Gherkin step glue generation for unimplemented steps, Gherkin step code completion, Step-to-glue method code hopping, Gherkin syntax highlighting in ".feature" files matching step regex, and other useful features for Cucumber development.

* *“File 🡪 Settings 🡪 Plugins 🡪 Marketplace 🡪 Search for Cucumber for Java” click on install and then restart the Intellij.*

(You may validate cucumber installation after installing the plugin by producing a file with the suffix “.feature")

Step-2: Add the maven dependency

Add the following dependency to your pom.xml to build the step definitions:

<dependency>  
 <groupId>io.cucumber</groupId>  
 <artifactId>cucumber-java</artifactId>  
 <version>6.9.1</version>  
 <scope>test</scope>  
</dependency>

**Feature File:**

Gherkin feature files are text files that contain Gherkin behavior scenarios. They use the “.feature” extension, they belong in src/test/resources in a Maven project since they are not Java source files. They should be structured into a logical packaging structure as well.

Feature: Codoid Cucumber JVM Blog documentation  
  
 Scenario: User visit the Codoid official page  
 When I Navigate to the Codoid page  
 Then I should see the official page of Codoid

**Step-Definitions:**

Step definition classes are Java classes that include Gherkin step implementation methods. Step def classes have variables, constructors, and methods much like standard Java classes. Regular expressions are used to "glue" steps to methods.

Steps from any step definition class in the project can be used in feature file situations. Step definitions should be placed in packages under src/test/java in a Maven project, and their class names should end in "Steps."

There are various modes for building the Step definition class file: one is to generate the steps file manually, and the other is to use an Intellij shortcut.

* Through Intellij Shortcut:
* Place the caret at a step in your .feature file and press *Alt+Enter.*

The list of suggested intention actions opens.

* Select Create step definition to create a definition only for one step, or select Create all step definitions to add definitions for all steps in a scenario.
* If your project already has definition files, the IDE will offer you to choose which one you wish to add the additional step definitions to. You have the option of selecting an existing file or creating a new one.
* A window will appear if you want to create a new step definition file. Choose a file name, a type (Java), and a location for the new file. Files containing step definitions should be kept in a separate package. It's possible that leaving them in the default package will result in an error.
* Manually creating the Step definition file:

* Create the package under the src/java/<PACKAGE\_NAME>,

for example; “*src/java/stepDefinitions*”.

* Add the step definition file under the src/java/<PACKAGE\_NAME>,

for example; “*src/java/stepDefinitions/CodoidSteps*”.

package stepDefinitions;  
  
import io.cucumber.java.en.Then;  
import io.cucumber.java.en.When;  
  
public class CodoidSteps {  
  
 @When("I Navigate to the Codoid page")  
 public void iNavigateToTheCodoidPage() {  
 // Steps to execute  
 }  
  
 @Then("I should see the official page of Codoid")  
 public void iShouldSeeTheOfficialPageOfCodoid() {  
 //Steps to execute  
 }  
}

* Do the following to ensure that the step definition file is being created in the same order as our feature file. Ascertain that the file containing step definitions is in a separate package. You won't be able to utilize navigation otherwise.
* Keep Ctrl pressed when hovering the mouse pointer over a step in a .feature file. Click it when it turns into a link to get to the step definition. (or) Place your cursor on a step and choose Navigate | Declaration or Usages from the main menu, or just press Ctrl+B.

**Hooks:**

Hooks, which are chunks of code that execute before or after each scenario, are supported by Cucumber. Using the methods **@Before** and **@After**, you may specify them wherever in your project or step definition levels.

Cucumber Hooks help us manage the coding workflow and decrease code repetition. We may call it an invisible phase that enables us to carry out our scenarios or testing.

**@Before** - Before hooks execute before each scenario's initial step. Some of the use cases of @Before hooks are as follows:

* The most typical use case is to initialize a web driver. Before starting the test, we must first initialize the driver.
* Establish database connections: At the start of the test, the application may request access to test data.
* To set browser cookies: The programmer may require the setting of cookies at times in order to fulfil a functioning aim.
* To set up test data: At the start of the test, the application may demand access to test data.

**@After** - Even if steps are unsuccessful, undefined, pending, or skipped, @After hooks run after the last step of each scenario. This is a widely used steps for tasks that must be completed after the original scenario has been run.

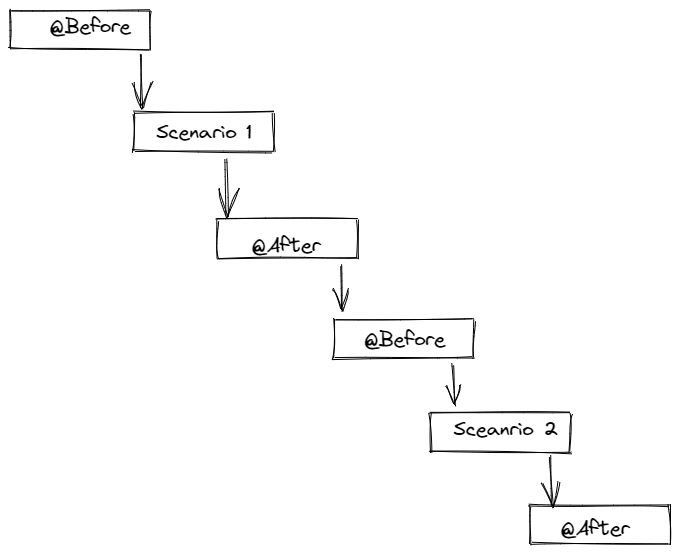
* Quit the web driver: This is the most usual method. To make testing independent, we're expected to terminate the browser after each test.
* Terminate DB connections: If the DB connection was created at the beginning of the test, it is best to terminate it at the conclusion.
* To delete the test data/browser cookies, follow these steps: We have cleanup activity here since the @After hook runs at the conclusion of each test.
* Log out of the application: Logging out of the application is critical for making testing self-contained.
* Capture screenshots for fail/pass scenarios: To ensure that the test runs properly, we must always take a screenshot in the event of a failure.

Example:

@Before  
public void initDriver() {  
 System.*out*.println("Open browser");  
 WebDriverManager.*chromedriver*().setup();  
 base.driver = new ChromeDriver();  
 base.driver.manage().window().maximize();  
 base.driver.manage().timeouts().implicitlyWait(10, TimeUnit.*SECONDS*);  
}

@After  
public void teardown() {  
 System.*out*.println("Close browser");  
 base.driver.quit();  
}

**Execution Order**: Before and After hooks are executed for each scenario.



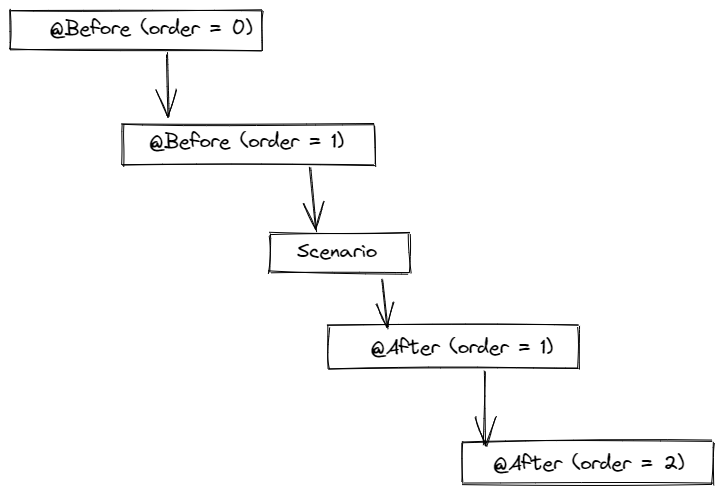
**Set the Priority in Cucumber Hooks**:

* Cucumber runs Hooks in a specific sequence, however there is a method to modify that order according on the test's requirements.
* @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, which means the opposite of @Before. Value 1 would run first and 0 would be after 1.

Example:

@Before(order = 0)  
public void initDriver() {  
 System.*out*.println("Open browser");  
 WebDriverManager.*chromedriver*().setup();  
 base.driver = new ChromeDriver();  
 base.driver.manage().window().maximize();  
 base.driver.manage().timeouts().implicitlyWait(10, TimeUnit.*SECONDS*);  
}  
  
@Before(order = 1)  
public void maximizeWindow() {  
 System.*out*.println("Maximize window");  
 base.driver.manage().window().maximize();  
}

**Execution Order of Hooks:**



**Hooks Tagging:**

* In the last part, we discussed how hooks are run before and after each scenario. The only difference between tagged hooks and regular hooks is that they are run before and after the given tag.
* If multiple situations need distinct requirements, then various hooks are required for each scenario. If we have two distinct tags, say smoke and regression, and we want to do various actions based on the tag, we may use tagged hooks to do so.
* Two instances are labelled with @Smoke and one with @Regression in the feature file below.

Example:

@After("@Regression,@Smoke")  
public void tearDown() {  
 System.*out*.println("Close browser");  
 base.driver.quit();  
}

**Cucumber Options:**

@CucumberOptions are similar to a property file or settings. It comes with a number of options that we utilize while dealing with the cucumber command line parameters.

Types of Cucumber Options, listed below

dryRun, features, glue, monochrome, plugin, strict, tags

**Sample:**

@CucumberOptions(features = {"src/test/resources/features/smoke"},  
 monochrome = true,  
 tags = {"@Smoke"},   
 glue = {"codoid"}, //locate the stepDefinition  
 plugin = {"pretty", "html:target/cucumber-report/",},  
 dryRun = true,  
 strict = true  
 )

**Features option:**

Identifies a feature file directory, a single feature, or specific scenarios. Cucumber will utilise the package of the annotated class if no feature path is supplied. If the annotated class is com.example.Runner, for example, features are presumed to reside in com.example.

**Glue Option:**

Glue code (step definitions, hooks, and plugins) is loaded from this package. Cucumber will utilise the package of the annotated class if no glue is given. If the annotated class is com.example.Runner, for example, glue is presumed to reside in com.example.

**dryRun option:**

dryRun to rapidly scan your features without having to run them. It ensures that every Step in the Step Definition file has the associated method. The dryRun option can be set to either true or false.

True: This means Cucumber will only verify whether or not each Step listed in the Feature File has corresponding code defined in the Step Definition file. So, if any of the functions in the Step Definition for any Step in the Feature File are missing, the notice will be shown. The code inside the step definition method will not run in this case.

When set to false, all statements/code included within the step definitions method will run.

**Monochrome option:**

True or false can be specified for this option. The console output for the Cucumber test will be much easier to comprehend if it's set to true. The console output will be less readable than it should be if it is set to false.

**Tags option:**

At the feature level, tags may also be defined. Once a tag is defined at the feature level, it is applied to all situations in that feature file. We can utilise more than one tag for a particular feature, depending on the nature of the case. A specified scenario will be run whenever Cucumber locates a suitable call.

***tags = {"@Smoke"} ->*** single tag

***tags = {"~@Smoke"} ->*** Ignore tag, none of the scenarios that include this tag will be evaluated.

***tags = {“@tag”, “@tag1”} ->*** means AND condition. –It says that scenarios matching both these tag needs to be executed.

***tags = {“@tag1, @tag2”} ->*** means OR condition. — It says that scenarios matching any of this tag needs to be executed.

**Strict option:**

If you use the —strict flag, cucumber will fail until all of the step definitions are provided.

false: If the strict option is false, then if cucumber detects any undefined/pending stages during execution, cucumber does not fail the execution and the undefined steps are skipped, resulting in an SUCCESSFUL BUILD.

true: If the Strict option is set to true, cucumber will fail the execution if it detects any undefined/pending steps during execution. Undefined steps will be marked as fail, and BUILD will be FAILURE.

**Plugin option**: Different formatting choices for the output reports are specified.

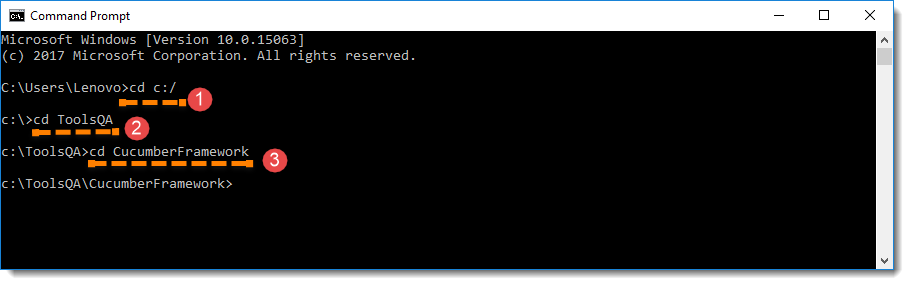
**Running Cucumber**:

There are different ways to run Cucumber Test

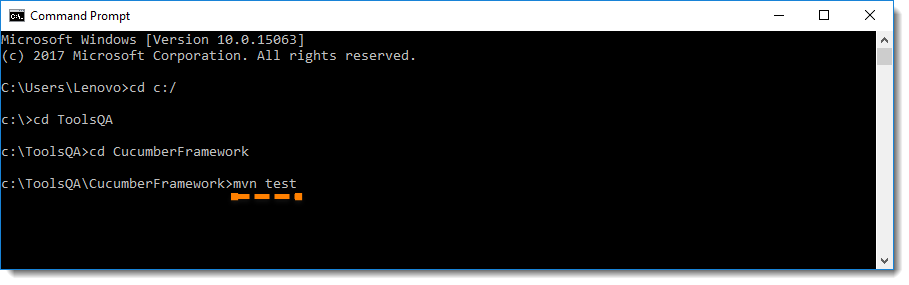
* command line
* IntelliJ IDEA or Eclipse
* Docker

**Run Test from Command Line (Using Maven):**

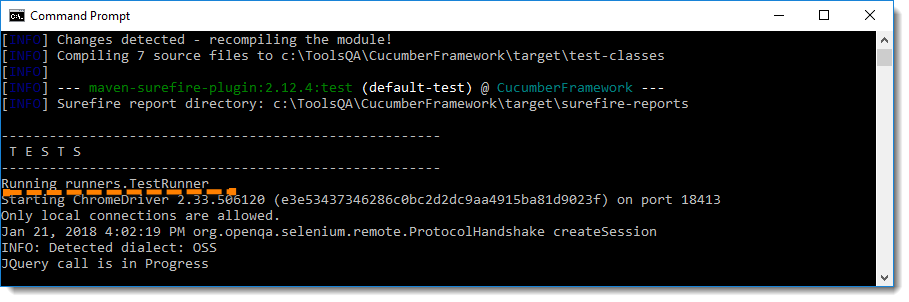
1. Open the **command prompt** and **cd** until the project root directory.



1. First, let's run all the Cucumber Scenarios from the command prompt. Since it's a Maven project and we have added Cucumber in **test scope** dependency and all features are also added in **src/test** packages, run the following command in the command prompt: **mvn test**



You would notice below that it actually triggered the **TestRunner** file.



**Build Success Output**

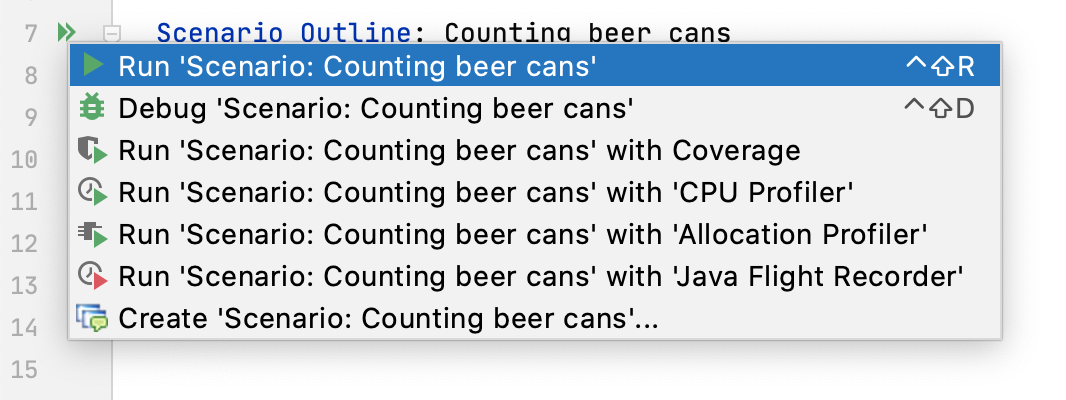


# **Run Cucumber tests ﻿ IntelliJ** **IDEA:**

The quickest way of running Cucumber tests is by using the icons in the gutter next to the necessary feature or scenario.

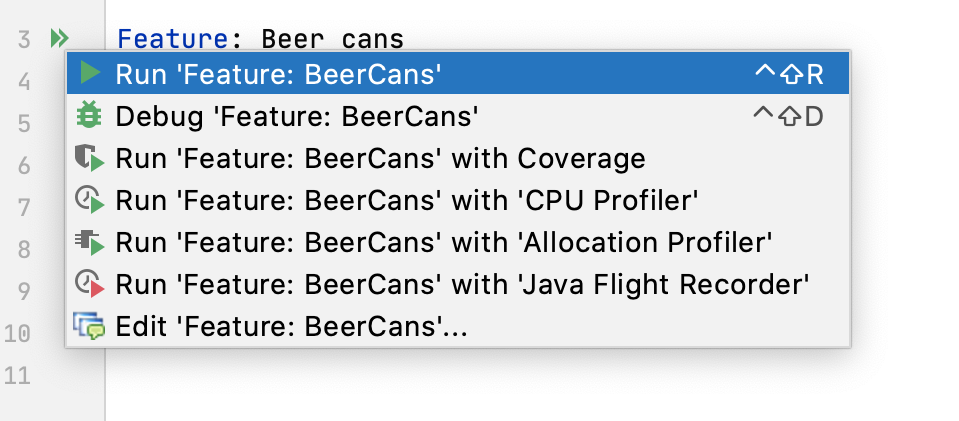
* **To run a scenario:**

Click in the gutter next to the scenario that you want to run and select **Run 'Scenario: <name>'**.

****

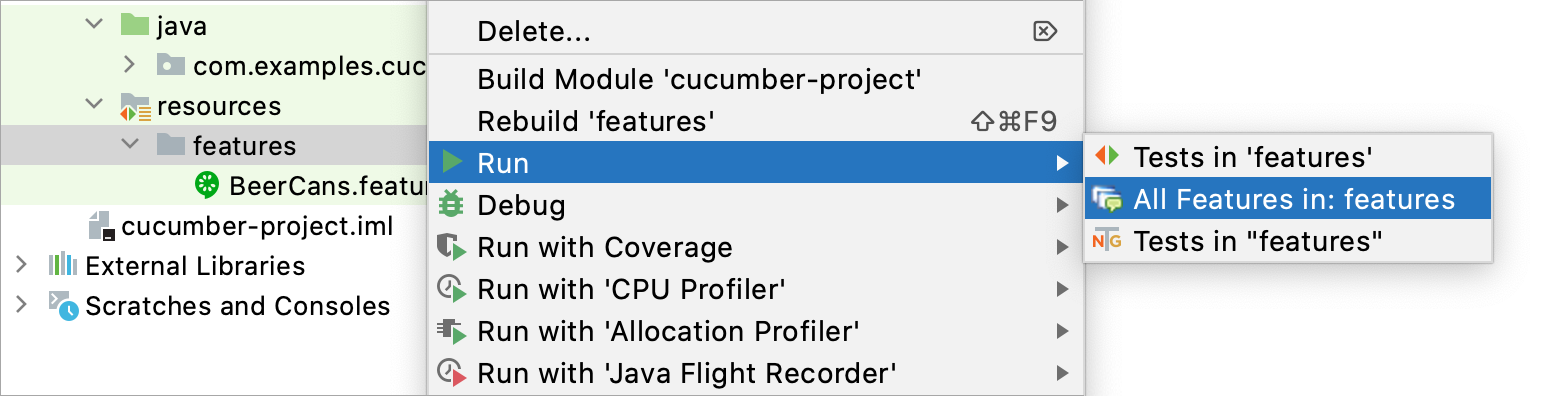
* **To run a feature:**

Click in the gutter next to the feature that you want to run and select **Run 'Feature: <name>'**.

****

* **To run all the feature file in the folder:**

In the **Project** tool window (Alt+1), right-click the **features** folder and select **Run all Features in: <directory name>**.

****

**Page Object:**

Page Object Model, also known as POM, is a design pattern in Selenium that creates an object repository for storing all web elements. It is useful in reducing 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.

**Advantages of Page Object Model**:

* **Helps with easy maintenance**: POM is useful when there is a change in a UI element or there is a change in an 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 the required changes in the right files. This makes test cases easy to maintain and reduces errors.
* **Helps with reusing code**: 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 easily identify actions that will be performed on a particular screen by navigating through the java file. If a change must be made to a certain section of code, 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 Example will create 2-page files.

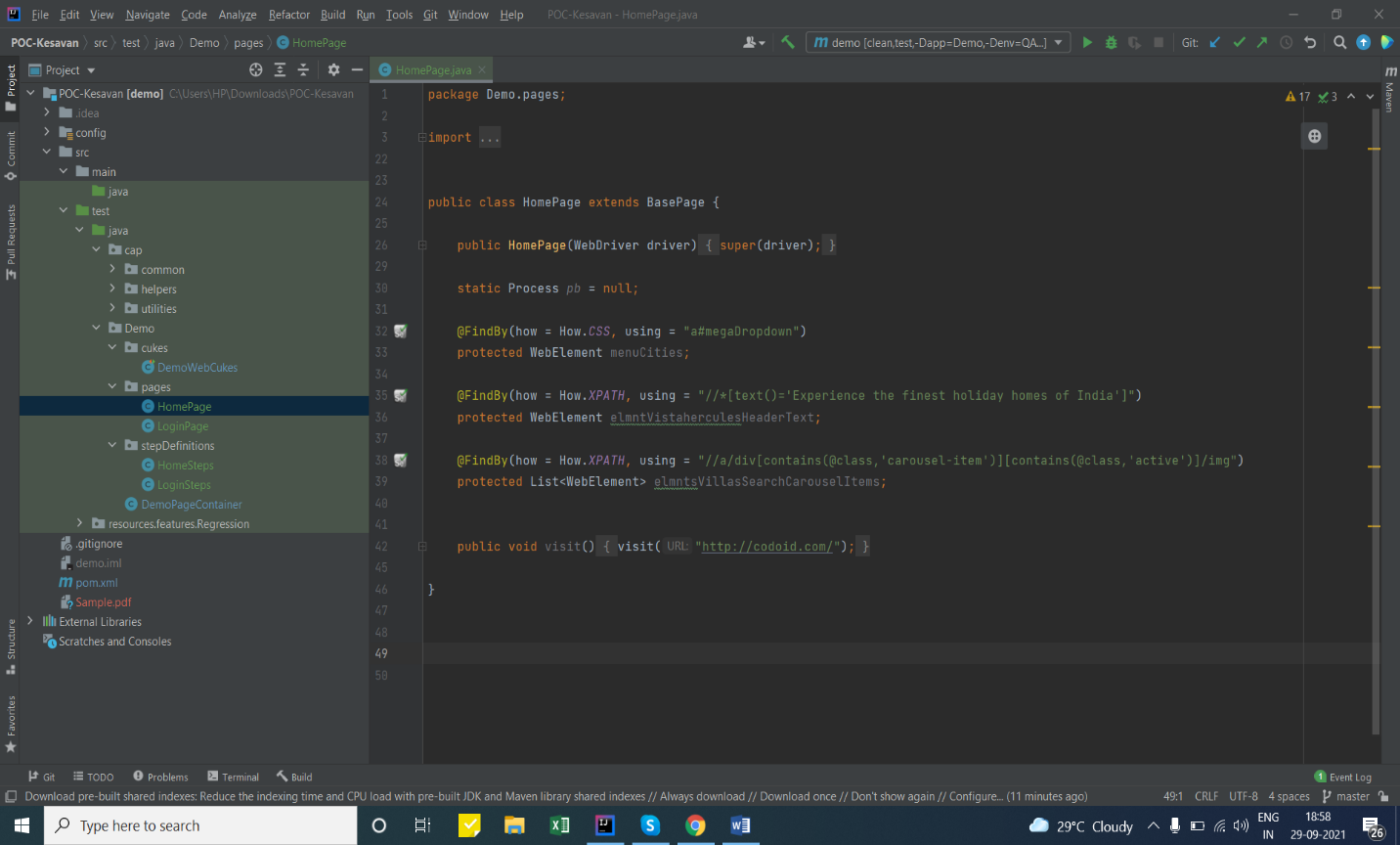
* HomePage
* LoginPage

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.

**Sample Project Structure for POM:**

### 

**HomePage Java File:**

****

**Explanation of Code**

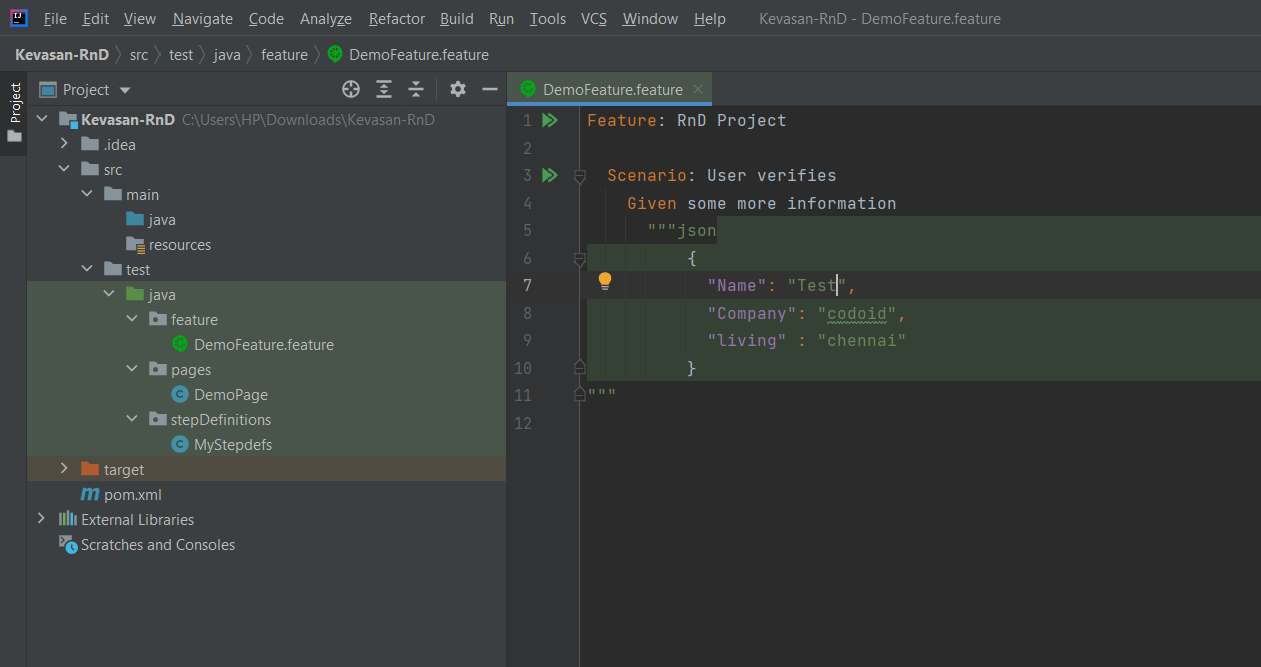
* **Code Line-32 to 39:** Identifying elements present on [**Home Page**](https://www.browserstack.com/)such as **Header Text** and **Menu** button
* **Code Line-42:** Performing actions on identified objects on Home Page

**New Features in Cucumber:**

1. **Doc Strings:**

**Doc Strings a**re handy for passing a larger piece of text to a step definition.

The text should be offset by delimiters consisting of three double-quote marks on lines of their own:



In your step definition, there’s no need to find this text and match it in your pattern. It will automatically be passed as the last argument in the step definition.

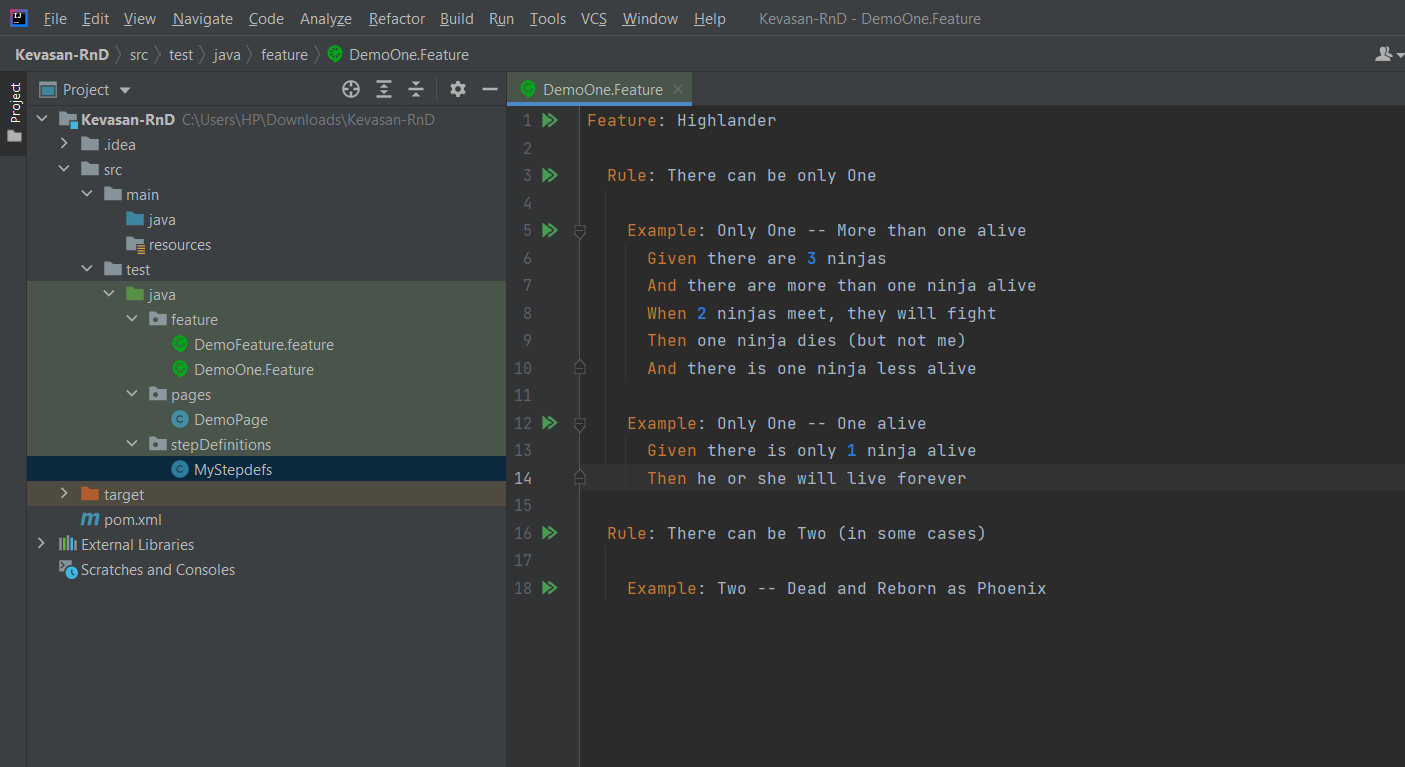
Indentation of the opening """ is unimportant, although common practice is two spaces in from the enclosing step. The indentation inside the triple quotes, however, is significant. Each line of the Doc String will be dedented according to the opening """. Indentation beyond the column of the opening """ will therefore be preserved.

1. **Rule:**

The (optional) Rule keyword has been part of Gherkin since v6.

The purpose of the Rule keyword is to represent one business rule that should be implemented. It provides additional information for a feature. A Rule is used to group together several scenarios that belong to this business rule. A Rule should contain one or more scenarios that illustrate the particular rule.

**For example:**

****

1. **Spoken Languages**

The language you choose for Gherkin should be the same language your users and domain experts use when they talk about the domain. Translating between two languages should be avoided.

This is why Gherkin has been translated to over 70 Languages.

**Cucumber Assertions:**

Assertions are used to perform various kinds of validations in the test cases, which in turn helps us to decide whether the test case has passed or failed. We consider a test as successful if it runs without any exception.

## **Types of Assertions In Selenium:**

There are two types of assertions in Selenium and the categorization depends on how the assertion behaves after a condition is pass or fail.

**Here, we would discuss two types of assertions in**[**Selenium**](https://www.softwaretestinghelp.com/selenium-tutorial-1/)**:**

* Hard Assertions
* Soft Assertions

**Hard Assertions:**

A hard assertion does not continue with execution until the assertion condition is met.

Hard assertions usually throw an Assertion Error whenever an assertion condition has not been met. The test case will be immediately marked as Failed when a hard assertion condition fails.

**Soft Assertions:**

A soft assertion continues with the next step of the test execution even if the assertion condition is not met.

Soft Assertions are the type of assertions that do not throw an exception automatically when an assertion fails unless it is asked for. This is useful if you are doing multiple validations in a form, out of which only a few validations directly have an impact on deciding the test case status.

## **Junit Assert Methods:**

**The various types of Junit Assert methods are explained below in detail.**

* 1. **assertEquals:** assertequals method compares the expected result with that of the actual result. It throws an AssertionError if the expected result does not match with that of the actual result and terminates the program execution at assert equals method.

**Syntax:**

public static void assertEquals(String expected, String actual)

* 1. **assertTrue:** asserttrue method asserts that a specified condition is true.

**Syntax:**

public static void assertTrue(java.lang.String message, boolean condition)

* 1. **assertFalse**: assert false method asserts that a specified condition is false.

**Syntax:**

public static void assertFalse(java.lang.String message, boolean condition)

**Parameters in Cucumber:**

We are going to pass parameters from Cucumber feature file to its corresponding steps file. What we are going to use:

1. IntelliJ as IDE
2. Selenium (for testing)
3. TestNG
4. Cucumber for TestNG dependency
5. Cucumber Java dependency

**Example:**

**Step 1:**

Let’s create a feature file for Cucumber steps. Make sure that Cucumber and Gherkin plugin is installed in IntelliJ. Create feature file “FirstTest.feature” and add following code to it.

Feature: Opening home page

 Scenario:

  Given I am on Home Page of <URL>

  Then I see Trivago's logo

Example:

| URL |

| <https://magazine.trivago.com/> |

In the above code, the parameter we are passing is the navigation URL which is in Greater than and lesser than symbol.

Now we are going to catch this parameter in our steps file.

**Step 2:**

Create a new java file “Steps.java” and add following code to it.

public class Steps {

    public static WebDriver driver;

    @Given("^I am on Home Page of \"([^\"]\*)\"$")

    public void i\_am\_on\_Home\_Page\_of(String arg1) throws Throwable {

        System.setProperty("webdriver.chrome.driver", "D:\\Selenium Webdriver/chromedriver.exe");

        driver = new ChromeDriver();

        driver.get(arg1);

    }

    @Then("^I see Trivago's logo$")

    public void i\_m\_taken\_to\_Contact\_Page() throws Throwable {    }

}

In the above snippet for Given statement, notice that we have

* **String arg1** in the class. This is where we are catching the sent parameter and utilizing it in the steps.
* **\”([^”]\*)\”**in the Given clause, which denotes that parameter is present at this point in the statement.

**Reporting in Cucumber:**

When we execute Cucumber Scenarios, it automatically generates an output in the eclipse and intelij console. There is a default behavior associated with that output and we can also configure that output as per our needs also. So how do we modify the default behavior, let's see this now.

### ****Pretty Report****

The first plugin, we will talk about is **Pretty**.  This provides more verbose output. To implement this, just specify plugin = "**pretty**" in CucumberOptions. This is what the code looks like:

* @CucumberOptions( plugin = { "pretty" } )

### ****Monochrome Mode Reporting****

If the monochrome option is set to false, then the console output is not as readable as it should be. The output when the monochrome option is set to false is shown in the above example. It is just because, if the monochrome is not defined in Cucumber Options, it takes it as ***false by default***. How to specify it:

* @CucumberOptions( monochrome = true );

You must be wondering that all we have seen above is actually good for a test or for a couple of tests. But if we run a full test suite, this report is not much useful in that case. On top of that, it is difficult to keep these console output safe for future use.

Cucumber gives us the capability to generate reports as well in the form of **HTML, XML, JSON & TXT**. Cucumber frameworks generate very good and detailed reports, which can be shared with all stakeholders. There are multiple options available for reports which can be used depending on the requirement.

### ****Cucumber HTML Reports:****

### For HTML reports, add **html:target/cucumber-reports** to the @CucumberOptions plugin option.

@CucumberOptions(

features = "src/test/resources/functionalTests",

glue= {"stepDefinitions"},

plugin = { "pretty", "html:target/cucumber-reports" },

monochrome = true

)

**Note**: We have specified the path of the Cucumber report, which we want it to generate it under the target folder.

This will generate an HTML report at the location mentioned in the formatter itself.

### ****Cucumber JSON Report****

For JSON reports, add **json:target/cucumber-reports/Cucumber.json** to the @CucumberOptions plugin option.

@CucumberOptions(

features = "src/test/resources/functionalTests",

glue= {"stepDefinitions"},

plugin = { "pretty", "json:target/cucumber-reports/Cucumber.json" },

monochrome = true

)

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

**Cucumber JUNIT XML Report**

For JUNIT reports, add **junit:targe/cucumber-reports/Cucumber.xml** to the @CucumberOptions plugin option.

@CucumberOptions(

features = "src/test/resources/functionalTests",

glue= {"stepDefinitions"},

plugin = { "pretty", "junit:target/cucumber-reports/Cucumber.xml" },

monochrome = true

)

**Note** : 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.

**We can even generate all reports together as well.**

@CucumberOptions(

features = "src/test/resources/functionalTests",

glue= {"stepDefinitions"},

plugin = { "pretty", "json:target/cucumber-reports/Cucumber.json",

"junit:target/cucumber-reports/Cucumber.xml",

"html:target/cucumber-reports"},

monochrome = true

)

**Dependency Injection:**

**Implementing Dependency Injection with Java-Selenium-Cucumber**

While coding an automation framework from scratch or on an existing one, you should take care of some basics. The framework should be maintainable, easy to understand, avoid coding duplicates, and quickly adapt to changes. To overcome these basics, you should follow some design principles and techniques in your framework. This article will show you one of these design patterns-Page Object Model (POM)-and explain its details.

**Dependency injection:**

Dependency injection (DI) is one of the design patterns and implementations of the IoC. DI takes the dependent object creation outside of the class and provides required objects in different ways. Using DI, you move the creation and binding of the dependent objects outside of the class that depends on them.

There are many DI types that you can use, such as Constructor Injection, Property Injection, and Method Injection. In this article, I will be talking about Constructor Injection.

**Constructor Injection**

In constructor injection, the injector provides the required dependency through the constructor. Some third-party tools would help you to implement DI on a constructor. I will be talking about Cucumber Picocontainer dependency, one of the third-party dependencies, to implement constructor injection.

**Cucumber Picocontainer**

Cucumber Picocontainer is a third-party framework for providing required dependencies in your classes. This framework will make your classes loosely coupled and take all responsibility for injecting required dependencies. You can use it in your framework to achieve loose coupling. Now let’s implement DI in the automation framework with Cucumber Picocontainer.

**Usage of Cucumber Picocontainer**

To be able to use Cucumber Picocontainer, you should add this dependency in your classpath. Make sure that you have added to the same version with JUnit and cucumber dependencies. Otherwise, you will get an exception while trying to use Picocontainer dependency.

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-picocontainer -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-picocontainer</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

After adding the above dependency, you should create a constructor in your page definition class and pass the required objects there.

public class HomePageStepDefinitions {

private Homepage homepage;

public HomePageStepDefinitions(Homepage homepage) {

this.homepage = homepage;

}

}

@Given("user should be able to login {string} {string}")

public void user\_should\_be\_able\_to\_login(String username, String password) {

homepage.login(username, password);

}

As you see in the example above, you are only responsible for creating constructors and reference names. The rest of the responsibility for providing objects to your class belongs to third-party dependency, Cucumber Picocontainer.

After implementation, you are only responsible for creating the constructor and defining what you need in the class. Picocontainer becomes the main responsible for the rest of the assigning and injection. If any upcoming changes are required, you will be responsible for making changes only on the constructor, and you won’t take care of the rest.

**Conclusion:**

I have explained how to implement Dependency Injection to your automation framework and use it in the steps class. I have covered constructor injection and its usage in this article.

Finally, we can say you can move your automation frameworks to one step forward with implementing design approaches like dependency injection.

**Limitation in Cucumber JVM:**