Основная трудность заключается в том чтобы корректно передать аргументы из feature файла в параметры метода реализации. С верху метода (step definition) указывается аннотация одноимённая с анотацией из feature файла(это сделано для лутшей читаемости человеку),в скобках у этой аннотации идёт описание того шага который имплементируется в данном методе из feature файла, за щот этого описания cucumber понимает какой шаг реализуется каким методом - glue code. Вот в этом описании, in documentation they call it **expression** которое делает сам cucumber, нужно грамотно отразить те аргументы которые имеются в этом шаге в feature файле.

So the feature fail is written in a Gherkin syntax, and it is mapped with the step definition with aid of cucumber expression (то самое описание метода одноименной аннотацией). If there are parameters in the scenario in the gherkin syntax then it will be converted to the method arguments.

Step definition expression (то самое описание метода) can either be a **regular expression** or **cucumber expression** . A few years ago it was only regular expressions. Now it is basically cucumber expressions.

If you want to use regular expressions you need to put a special sign ‘^’ at the beginning of the expression and ‘$’ at the very end of expression, in that case cucumber will know that you are trying to use regExp. For example: “^I enter the **([^\"]\*)** and **([^\"]\*)$“** <- - but this is hard to understand for human so **cucumber expressions** was invented

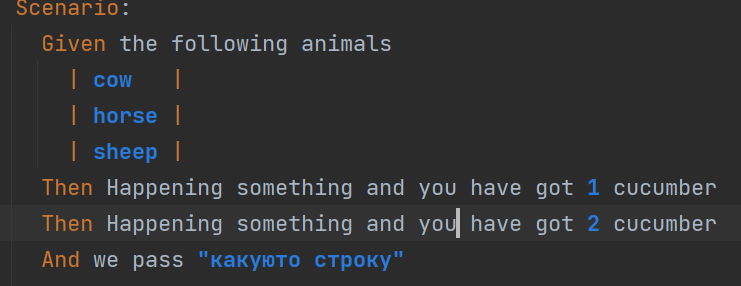
**cucumber expressions**

<https://github.com/cucumber/cucumber-expressions#readme> <- - link for docks of cucumber expressions

**Optional text**

It's grammatically incorrect to say *1 cucumbers*, so we should make the plural / множественное число s optional. That can be done by surrounding the optional text with parentheses. We should be able to use the same step definition method even if we have got two different steps in feature. Let say one step says one cucumber and the other step says two cucumbers. But since the logic is the same, both steps should match with the same step definition. So this is what is supported by using the optional text. **It helps in reusing the step definition,** and that what is aure target should be –Reusing the step definitions as much as possible.

For example, below are 2 steps.



Step definition looks like this:

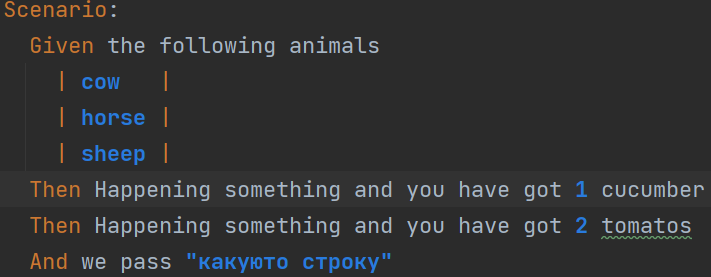
@Then("Happening something and you have got {int} cucumber(s)") <- - there is plural (s)

public void happeningSomethingAndYouHaveGotCucumber(int arg0) {

One step definition for both steps.

**Alternative text**

The idea for that option is Reusing the step definitions as much as possible.

If the steps different slightly than we can use ‘/’ but no space. For example we have 2 steps with only one word different. Here 2 steps Then and logically they are the same so in step definition we can use ‘/’ and that gives an opportunity to have only one step definition

@Then("Happening something and you have got {int} cucumber(s)**/**tomatos") <- - It is because we use ‘/’

**DEFAULT PARAMETERS TYPES**

**int**

Cамое простое это отрозить int, cucumber сам понимает как это сделать

Пример:

Then Happening something and you have got **2** ← step from feature fail with the integer in it

@Then("Happening something and you have got {int}") ← step definition, with {int} than you can pass it to parameters

**String**

And we pass **"какуюто строку"** < ← step from feature fail with String in it

@And("we pass {string}") ←step definition, with {string} than you can pass it to parameters

**Collections**

The simplest way to pass a List<String> to a step definition is to use a data table:

Given the following animals:

| **cow** |

| **horse** |

| **sheep** |

@Given("the following animals:") ←cucumber сам подставляет такое выражение с двоеточием врезультате

public void the\_following\_animals(**List<String> animals**) {//надо только подставить аргументы

}

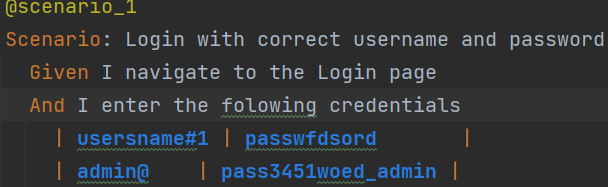
In this case, the DataTable is automatically flattened to a List<String> by Cucumber (using DataTable.asList(String.class)) before invoking the step definition.

All step definitions are loaded (and defined) before Cucumber starts to execute the plain text in the feature file. Once execution begins, for each step, Cucumber will look for a registered step definition with a matching Regexp. If it finds one, it will execute it, passing all capture group s and variables from the Regexp as arguments to the method.

Работа с таблицами

в общем бывают таблицы вот такие

**1)**

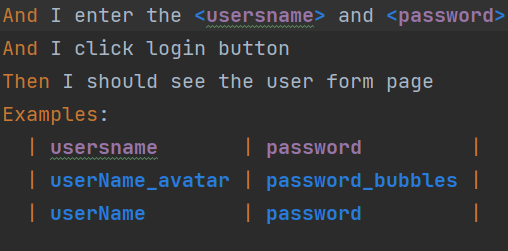


реализация вот такая

@And("I enter the folowing credentials")

public void iEnterTheFolowingCredentials(**DataTable** dataTable) {...}

В теле метода обращаемся к **DataTable** она хранит все элементы из таблицы в feature файла. Т.е. в описании к методу ничего делать не надо, а вот в параметрах метода уже указываем **DataTable.**

**2)** 

реализация вот такая:

@And("^I enter the **([^\"]\*)** and **([^\"]\*)$**")

public void iEnterTheUsersnamePassword(String usern, String passw) {...}

Здесь надо обратить внимание на регулярные выражения в описании к методу, этот приём используется в scenario outline, именно они отвечают за динамическое подставление аргументов из feature файла. Всё это хуйня. В данном примере используются регулярки, это сложно, у cucumber есть свои cucumber expression. Заключаются в фигурные скобки

@And(“I enter the {string} and {string}”) <- - намного проще.

**CUSTOM PARAMETERS TYPE**

In cucumber it is possible to create your own objects (custom data types) by converting the output parameters to your own types (by saying outputs it means the output from cucumber expression (то самое описание метода одноименной аннотацией, в котором могут быть параметры которые подтягиваются и feature файла на основе которых и можно создать обьект).

For example we have step definition

@And("we pass {string}") <- - we can see some String parameter in cucumber expression, but we

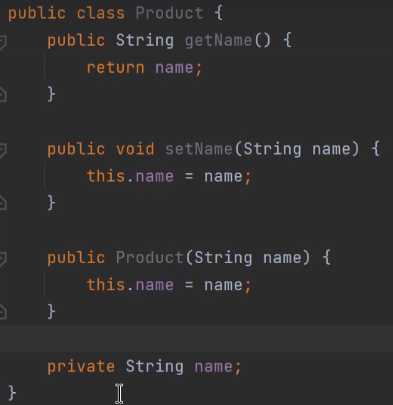
public void wePass(String arg0) {...} // don't know what kind of information gives it as this String

It is not informative. To understand the logic of the parameters you have to look at the feature fail. We can change the name of arguments to make it more clear, for example:

@And("we pass {string}")

public void wePass(String productName) {...} <- - in that case it makes more sense.

Now for example I want to share this productName with multiple step definitions. In that case you might want to create a product object to be able to share it. This object is called a **domain object** (domain - сфера, область.<https://docs.google.com/document/d/1EXkctFmrL9NuFshML1jXj9agPkeuzOVnYdZ4ZZ3PDII/edit>) It is advisable to use **domain objects**(objects that is related to the domain we are working on) while using a cucumber, because we are dealing with domain specific language - the Gherkin. The domain objects can be: product, address, credit card, debit card. So it makes sense to work with the domain objects for our website since it is an e-commerce website and use those objects among the steps. It also adds readability by looking at the step definition. It is easy to say which domain object you are dealing with, because it is readable.

So to create an object you need to use that productName String that comes from cucumber expression and then goes to method argument. Let's do that. We can create a package and name it for example objects. Then let's create a class Product. 

This is an example of POJO class.

Now we can create object in step definition using the String that came from cucumber expression.

@And("we pass {string}")

public void wePass(String productName) {

Product product=new Product(productName);

}

But after doing so we still didn't get more readability to cucumber expressions, and we need to write code in step definition. So there is one more way to create objects in cucumber.

@And("we pass {product}") <- - now it is very easy to read, and the product is a predefined keyword.

public void wePass(Product product) {

System.out.println(product.getName());

}

1)-We put ‘product’ - to the cucumber expression

2)-we are passing the Product as argument to the method

3)- We need to create a **method** annotated with @parameterType annotation that mthod will create an object. The name of that method should be the word that we are putting in curly braces (product) in cucumber expression. Then in annotation **@ParameterType**(".\*") to the method (lets call it - parameter type method) that creates the object in a braces we need to explain cucumber by using regular expression what type parameter should be and then use this parameter as an argument for the method that creates the object.

So altogether it will look like this:

@ParameterType(".\*") // annotation with regular expression

public Product product(String name){ // Name of the method + argument that came in the regular expression

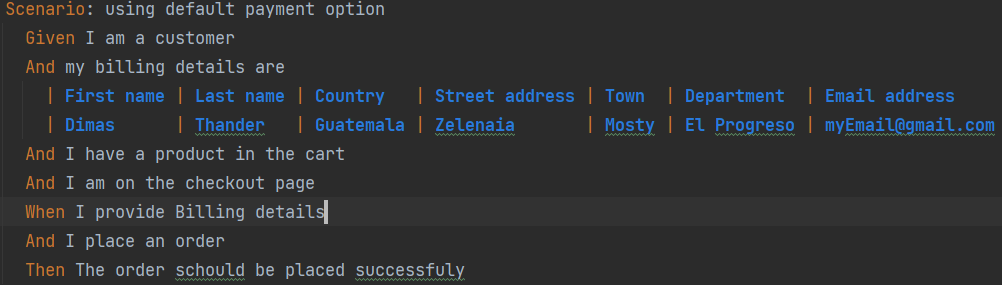
return new Product(name); // method should return object

}

And it might be a good idea to put the parameter type method in a separate package to a specific class. But make sure that you keep it together with step definitions in the same package, otherwise it's gonna be ekseption.

Also we can use @DataTableType <- - we can use this annotation as well in parameter type method, to work with tables

One more example:



we have a scenario, and in step ‘And’ we have multiple (data table) arguments that we need to use in step definition. Instead of passing all these arguments to one step definition (maybe we will want to use them in another step definition method as well), we can create a class POJO that will represent our domain object and we will be able to use the fields of the object in some different method in this state definition class. But again if we create object in a step definition class we have to pass all these arguments and inspite that in these case we will be able to use all the fields of our domain object in different methods it still we be a lot of code since we have to accept all the parameters from the scenario and initialize our object.

And here we can use custom parameters. We create package and class that we call for example

CystomDataTableType Then we create parameter type method, in arguments of that method cucumber may use kollektions (automatically) and the return type is our domain object. Important part is the annotation of these method it tells the cucumber the purpose of that method

@DataTableType

public BillingDitails billingDitailsInit(Map<String,String> map){

return new BillingDitails(map.get("First name"),

map.get("Last name"),

map.get("Country"),

map.get("Street address"),

map.get("Town"),

map.get("Department"),

map.get("Email address"));

}

By doing so we reduce the amount of code in our step definition file. Now we can create objects in step definition with one line of code that is easy to read.

private BillingDitails billingDitails;

@And("my billing details are")

public void myBillingDetailsAre(BillingDitails billingDitails) {

this.billingDitails=billingDitails;

}

So using the custom parameter types are good practice to reduce the amount of code in step definition, and to add readability. We simply move the realization to different place.