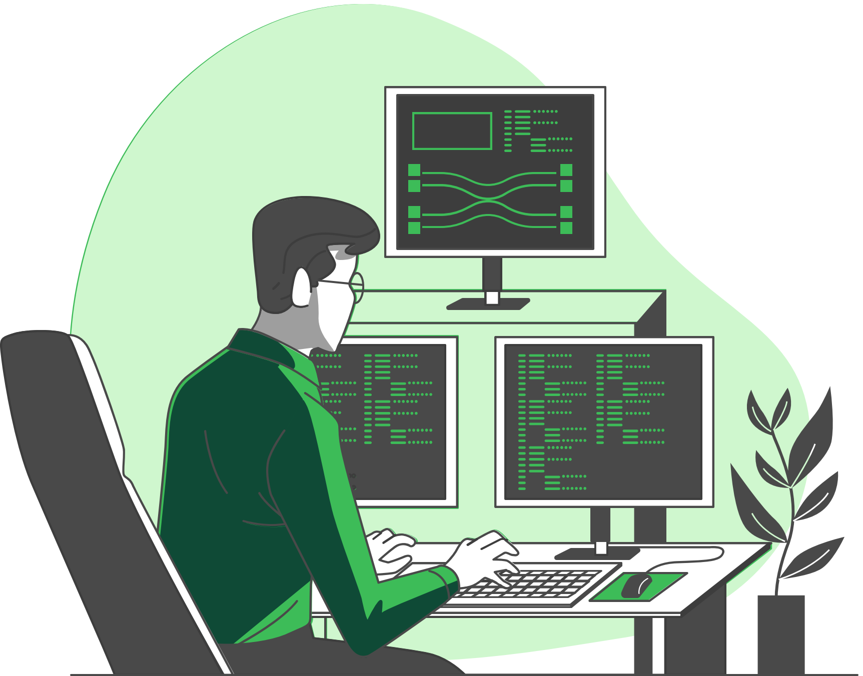
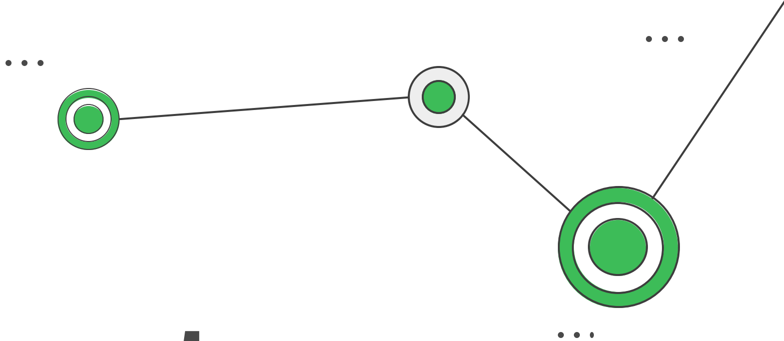




Acceptance Testing Workshop

SENG302

Learning Objectives

01

Appreciate

Understand the value of ATDD (acceptance test driven development) and how to get most out of this practice.

02

Gherkin

Learn how to write scenarios using Gherkin to capture the requirements accurately.

03

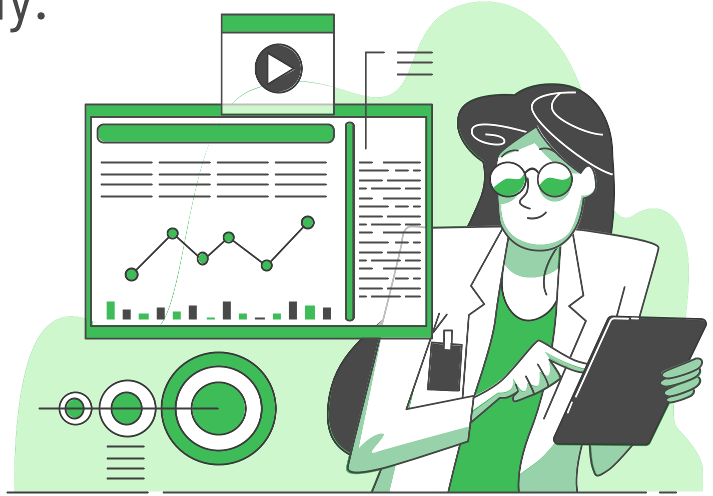
Automate

Be able to automate a given scenario using Cucumber test automation tool.

04

Apply

Practice these skills on your own project.



ATDD (Acceptance Test Driven Development)

- Involves team members with **different perspectives** (customer, development, testing) collaborating to write acceptance tests in **advance** of implementing the corresponding functionality. - *AgileAlliance*

- Used interchangeably with **Story Test Driven Development** (SDD), **Specification by Example** and **Behaviour driven development** (BDD)

- The essence of the all of the above approaches is having **conversations with various stakeholders** to ensure **shared understanding** of what is required.

"The hardest single part of building a software system is deciding precisely what to build"

— No Silver Bullet, Fred Brooks

Why do we need acceptance tests/examples/scenarios?

- It is very difficult to figure out **exactly what PO wants** you to build.
- What often happens in industry is because of a **misunderstanding**, code that has been worked on for several days or more had to be thrown away.
- Have you experienced this so far?

Acceptance Tests – How to Write Them

01 Use the language and terminology that **makes sense** to our stakeholders

02 Use **concrete, real-life** examples

03 Stakeholders should be able to really **imagine** themselves **using the system**

04 Use **Gherkin** structure to formulate tests

Acceptance Tests – Process Benefits

- They **facilitate conversation** with the PO during planning

- They **simplify** and **improve** the **accuracy of estimation** as the developers have a clearer idea of what they are expected to implement

- They provide a good guide for determining how to **split large stories**

- They act as a project **specification** and **documentation**

Acceptance Test Example

- Let's imagine you're building a credit card payment system. One of the requirements or acceptance criteria is to make sure users can't enter bad data.

Here's one way of expressing that:

"Customers should be prevented from entering invalid credit card details."

Acceptance Test Example Continues

- This requirement is useful, but it leaves far too much room for ambiguity and misunderstanding
- It lacks precision

Questions we would need to ask:

- What exactly makes a set of details invalid?
- How exactly should the user be prevented from entering them?

…

Acceptance Test Example Continues

- Let's try to illustrate this requirement with a concrete example:

“If a customer enters a credit card number that isn't exactly 16 digits long, when they try to submit the form, it should be redisplayed with an error message advising them of the correct number of digits.”

Gherkin

Gherkin

- Gherkin is a plain-text language with a lightweight structure for documenting examples of the software's behavior
- Gherkin is the language that **Cucumber** (acceptance tests automation tool) understands
- It uses **GIVEN-WHEN-THEN** format where:

GIVEN: A set of initial circumstances
WHEN: Some event happens
THEN: The expected result as per the defined behavior of the system

Gherkin – Special Keywords

- Gherkin files use the **.feature** file extension.
- They're saved as plain text, so they can be read and edited with simple tools.
- A Gherkin file is given its **structure** and **meaning** using a set of **special keywords**.

Gherkin – Special Keywords

Features

Background

Scenario

Given

When

Then

And

But

***

Scenario Outline

Examples

Feature

- Each Gherkin file begins with **Feature** keyword
- It gives you a convenient place to put some **summary documentation** about the **group of tests** that follow
- You can use this place to write **details** about **who** will use the feature, and **why**, or to put links to supporting documentation.
- Be **consistent** with feature file naming
- User logs would be stored in ***user_logs_in.feature*** or ***UserLogsIn.feature***
- Template for describing feature:

In order to <*meet some goal*>
As a <*type of stakeholder*>
I want <*a feature*>

by Chris Matt and Liz Keogh

Scenario

- To express the behavior we want, each feature contains **several** scenarios.
- Each scenario is a single concrete example of how the system should behave in a particular situation.
- Each feature typically has somewhere between five and twenty scenarios, each describing **different examples** of how that feature should **behave** in a different **circumstances**.
- Scenarios explore **edge cases** and **different paths** through a feature.
- Scenarios all follow the same pattern:

1. Get system into certain state
2. Poke it
3. Examine the new state

Scenario Continues

- We use **Given**, **When**, **Then** to identify those three different parts of the scenario.
- Each of the lines in a scenario is known as a step.
- We can add more steps to each **Given**, **When** or **Then** section of the scenario using keywords **And** and **But**.
- Cucumber does not care which of these keywords you use; the choice is simple there to help you create the most readable scenario. If you do not want to use **And** or **But**, you could create scenario by repeating **Given**, **When** and **Then** as many times as you need to in the context.

Scenario Readability – Different Keywords

Background:

    Given I have chosen some items to buy
    Given I am about to enter my credit card details

Scenario: Credit card number too short

    When I enter a card number that's only 15 digits long
    When all the other details are correct
    When I submit the form
    Then the form should be redisplayed
    Then I should see a message advising me of the correct number of digits

Scenario Readability – Different Keywords

Background:
 Given I have chosen some items to buy
 Given I am about to enter my credit card details

Scenario: Credit card number too short
 When I enter a card number that's only 15 digits long
 And ~~**When**~~ all the other details are correct
 And ~~**When**~~ I submit the form
 Then the form should be redisplayed
 And ~~**Then**~~ I should see a message advising me of the correct number of digits

We could have used **And** for second and third **When** and second **Then**

A Scenario is Stateless

- Each scenario must **make sense** and be able to be **executed independently** of any other scenario.
- When writing scenario, always assume that it will run against the system in a **default, blank state**.
- Tell the story from the beginning, using Given steps to **set up all the state** you need for that particular scenario.

Scenario Names

- Choose a scenario name carefully as it is important to get it right as:
    - When your tests break, it is the falling scenario's name that will be **reported** ...
    - Once you have quite a few scenarios in a feature file you do **not want to read steps** in order to know what is the scenario about.
    - As your system evolves, your stakeholders will quite often ask you to **change the expected behaviour** in an existing scenario. This could be another And line in a well-composed scenario.
    - Avoid putting anything about the outcome(the Then part) of the scenario into the name, concentrate on **summarizing** the context and event (Given and When)
- It's valid Gherkin to follow the name with a multi-line description-everything up until the first **Given** will be slurped up into the description of the scenario.

Scenario Readability

- When writing scenarios, readability is your main goal.
- A reader can easily feel more like they're reading a computer program than a specification document, which is what we want to try to avoid at all costs.
- There are few things we can do to improve **readability** of our feature files:

Background

Data Tables

Tags and Subfolders

Scenario Outline

Scenario Readability - Background

- Background section in a feature file allows you to specify a set of steps that **are common to every scenario in the file**.
    - If you ever need to change those steps, you have to change them in only **one place**.
    - The importance of those steps fades into the background so that when you're reading each individual scenario, you can focus on what is unique and important about that scenario.

Scenario Readability – Background Example

Feature: Change PIN
As soon as the bank issues new cards to customers, they are supplied with a Personal Identification Number (PIN) that is randomly generated by the system. In order to be able to change it to something they can easily remember, customers with new bank cards need to be able to change their PIN using the ATM.

Background:
Given I have been issued a new card
And I insert the card, entering the correct PIN
And I choose “Change PIN” from the menu

Scenario: Change PIN successfully
When I change the PIN to 9876
Then the system should remember my PIN is now 9876

Scenario: Try to change PIN to the same as before
When I try to change the PIN to the original PIN number
Then I should see a warning message
And the system should not have changed my PIN

Scenario Readability – Data Tables

- Sometimes steps in a scenario need to describe data that doesn't easily fit on a single line of Given, When, or Then. Gherkin allows you to place these details in a table right underneath a step.

Instead of writing the following:

Given a user "Mike Johns" born on August 29, 1965
And a User "Sam Picalo" born on January 8, 1935
And a User " Maria Ferguson" born on October 9, 1940
.....

We can write this:

Given these Users:
|name |date of birth
|Mike Johns |August 29, 1965
| Sam Picalo |January 8, 1935
| Maria Ferguson|October 9, 1940

Scenario Readability – Data Tables

Feature:

Scenario:

Given a board like this:

	1	2	3
1			
2			
3			

When player X plays in row 2, column 1

Then the board should look like this:

	1	2	3
1			
2	X		
3			

Scenario Readability – Scenario Outline

- Sometimes you have **several scenarios** that follow exactly the **same** pattern of steps, just with different values or expected outcomes. All the repetition in the feature you will end up with will make it boring to read.
- Scenario outline example:

Feature: Withdraw Fixed Amount

The "Withdraw Cash" amount contains several fixed amounts to speed up transactions for users.

Scenario Outline: Withdraw fixed amount

Given I have <Balance> in my account

When I choose to withdraw the fixed amount of <Withdrawal>

Then I should receive <Received> cash

And the balance of my account should be <Remaining>

And <Outcome> message should displayed

Scenario Readability – Scenario Outline Continued

Examples:

Balance	Withdrawal	Received	Remaining	Outcome
$500	$50	$50	$450	received $50 cash
$500	$100	$100	$400	received $100 cash
$100	$200	$0	$100	Not enough funds

Scenario readability – tags and subfolders

- As your test suit starts to grow, you'll want to keep things tidy so that the documentation is easy to **read** and **navigate**.
- **Subfolders** - think about your features as a book that describes what your system does, then the subfolders are like the chapters in that book.

features/

reading_data_from_mock/

reading_data_from_livestream

- **Tags** – if subfolders are the chapters in your book of features, then tags are sticky notes you've put on pages you want to be able to find easily.
- You tag a scenario by putting a word with the @character on the line before the Scenario keyword, like this:

@story45

Scenario: Estimate time to the next mark

Given I am sailing at speed 50 knots /h

And I am 3 miles away from the next mark

Then my estimated arrival time should be 3 min

You can have many tags per scenario

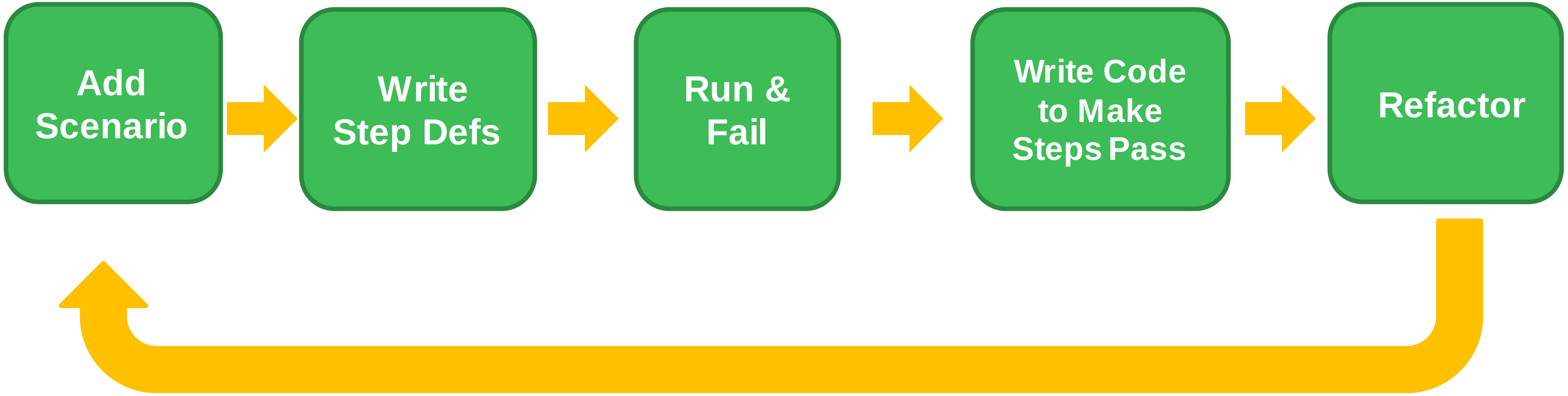
How do we automate scenario testing?

Using Cucumber to Automate Scenario Testing

- Cucumber is a testing framework, driven by plain English text - collaboration tool
- Helps to execute plain text functional description as automated tests
- Outside-in approach, where programmers incrementally write and run the scenarios using Cucumber until the feature passes all the tests
- Supports about 60+ languages
- Automation can be written in our favourite language (Ruby, Java, C#, Javascript, Scala, Groovy, Python, Perl C++ etc)

Life Cycle

- Resembles TDD
- All scenarios should be written FIRST (during or soon after Planning 1 & 2)
- Like a contract with the PO about what will be implemented during the sprint



Step Definitions

- A step definition is a Java method with an expression that links it to one or more Gherkin steps
- Step definitions aren't linked to a particular feature file or scenario.
- The file, class or package name of a step definition does not affect what Gherkin steps it will match. The only thing that matters is the step definition's expression.
- Running Cucumber Test Runner class generates a step definition skeleton for existing scenario(s) with no matching step definitions.
- By convention you will store step definition package under test/java package.

Two ways of writing step definition expressions

- A Gherkin step:

  "Given I'm logged in as a guest"

  can be written in any of the following ways:

- With cucumber expression

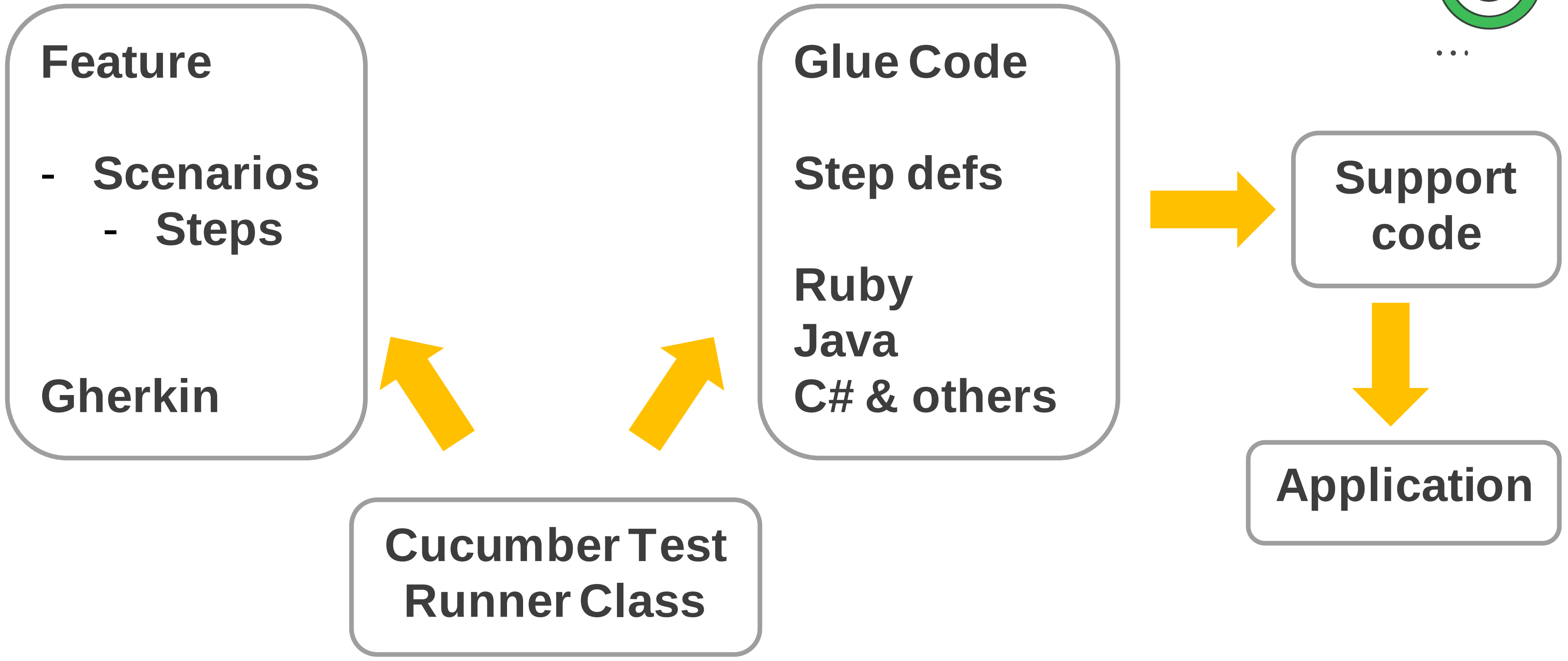
```
@Given("I'm logged in as a(n) {string}")
public void ImLoggedInAsA(string role)
{// log in as the given role}
```

- With regular expression

```
@Given("I'm logged in as a(n) (.*)")
public void ImLoggedInAsA(string role)
{// log in as the given role}
```

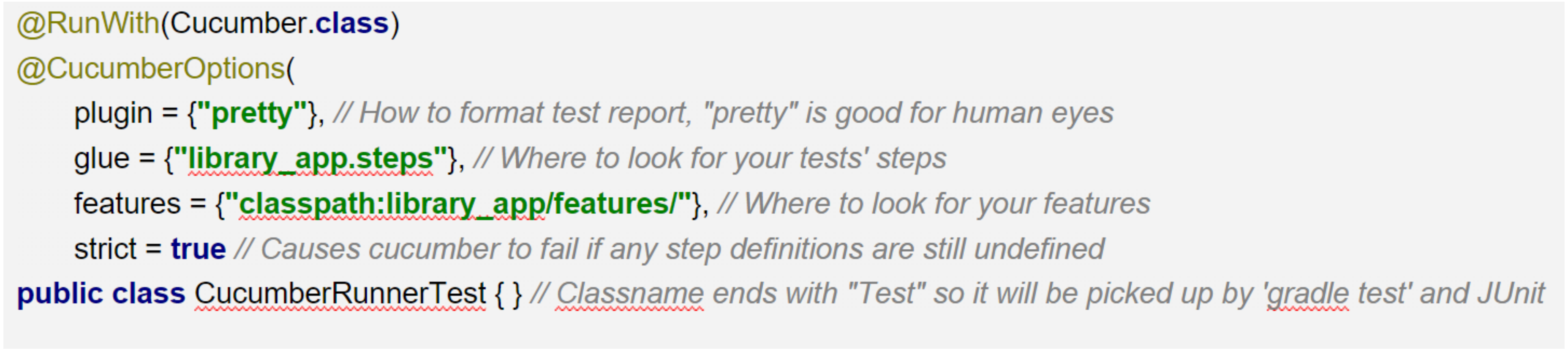
We will be using Cucumber expressions mostly for readability and simplicity

Cucumber Design



Cucumber Test Runner

- Runs a cucumber feature file
- Acts as an interlink between feature files and step definition classes
- Picked up by ./gradlew test as if it were a JUnit test class



```java
@RunWith(Cucumber.class)
@CucumberOptions(
      plugin = {"pretty"}, // How to format test report, "pretty" is good for human eyes
      glue = {"library_app.steps"}, // Where to look for your tests' steps
      features = {"classpath:library_app/features/"}, // Where to look for your features
      strict = true // Causes cucumber to fail if any step definitions are still undefined
public class CucumberRunnerTest { } // Classname ends with "Test" so it will be picked up by 'gradle test' and JUnit
```

Returning Results

- A scenario that's been executed can end up in any of the following states:
- Failing – If block of code executed raises an exception
- Pending – Cucumber finds a step definition that's not fully implemented
- Undefined – Cucumber can not find a step definition that matches a step
- Skipped – When one step in a scenario fails the rest of the steps will be skipped
- Passed – all the steps in the scenario have been successfully executed

Let's apply what we have learned!