Reference

Gherkin uses a set of special keywords to give structure and meaning to executable specifications. Each keyword is translated to many spoken languages; in this reference we'll use English.

Most lines in a Gherkin document start with one of the keywords.

Comments are only permitted at the start of a new line, anywhere in the feature file. They begin with zero or more spaces, followed by a hash sign (`#`) and some text.

Block comments are currently not supported by Gherkin.

Either spaces or tabs may be used for indentation. The recommended indentation level is two spaces. Here is an example:

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

The trailing portion (after the keyword) of each step is matched to a code block, called a step definition.

Please note that some keywords *are* followed by a colon (`:`) and some *are not*. If you add a colon after a keyword that should not be followed by one, your test(s) will be ignored.

Keywords

Each line that isn't a blank line has to start with a Gherkin *keyword*, followed by any text you like. The only exceptions are the free-form descriptions placed underneath `Example` / `Scenario`, `Background`, `Scenario Outline` and `Rule` lines.

The primary keywords are:

- `Feature`
- `Rule` (as of Gherkin 6)
- `Example` (or `Scenario`)
- `Given`, `When`, `Then`, `And`, `But` for steps (or `*`)
- `Background`
- `Scenario Outline` (or `Scenario Template`)
- `Examples` (or `Scenarios`)

There are a few secondary keywords as well:

- `"""` (Doc Strings)
- `|` (Data Tables)
- `@` (Tags)
- `#` (Comments)

> **LOCALISATION**
>
> Gherkin is localised for many spoken languages; each has their own localised equivalent of these keywords.

Feature

The purpose of the `Feature` keyword is to provide a high-level description of a software feature, and to group related scenarios.

The first primary keyword in a Gherkin document must always be `Feature`, followed by a `:` and a short text that describes the feature.

You can add free-form text underneath `Feature` to add more description.

These description lines are ignored by Cucumber at runtime, but are available for reporting

(they are included by reporting tools like the official HTML formatter).

```
Feature: Guess the word

  The word guess game is a turn-based game for two players.
  The Maker makes a word for the Breaker to guess. The game
  is over when the Breaker guesses the Maker's word.

  Example: Maker starts a game
```

The name and the optional description have no special meaning to Cucumber. Their purpose is to provide a place for you to document important aspects of the feature, such as a brief explanation and a list of business rules (general acceptance criteria).

The free format description for `Feature` ends when you start a line with the keyword `Background`, `Rule`, `Example` or `Scenario Outline` (or their alias keywords).

You can place tags above `Feature` to group related features, independent of your file and directory structure.

You can only have a single `Feature` in a `.feature` file.

Descriptions

Free-form descriptions (as described above for `Feature`) can also be placed underneath `Example`/`Scenario`, `Background`, `Scenario Outline` and `Rule`.

You can write anything you like, as long as no line starts with a keyword.

Descriptions can be in the form of Markdown - formatters including the official HTML formatter support this.

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:

```
# -- FILE: features/gherkin.rule_example.feature
Feature: Highlander

  Rule: There can be only One

    Example: Only One -- More than one alive
      Given there are 3 ninjas
      And there are more than one ninja alive
      When 2 ninjas meet, they will fight
      Then one ninja dies (but not me)
      And there is one ninja less alive

    Example: Only One -- One alive
      Given there is only 1 ninja alive
      Then they will live forever ;-)

  Rule: There can be Two (in some cases)

    Example: Two -- Dead and Reborn as Phoenix
      ...
```

Example

This is a *concrete example* that *illustrates* a business rule. It consists of a list of steps.

The keyword `Scenario` is a synonym of the keyword `Example`.

You can have as many steps as you like, but we recommend 3-5 steps per example. Having too many steps will cause the example to lose its expressive power as a specification and documentation.

In addition to being a specification and documentation, an example is also a *test*. As a whole, your examples are an *executable specification* of the system.

Examples follow this same pattern:

- Describe an initial context (`Given` steps)

- Describe an event (`When` steps)
- Describe an expected outcome (`Then` steps)

Steps

Each step starts with `Given`, `When`, `Then`, `And`, or `But`.

Cucumber executes each step in a scenario one at a time, in the sequence you've written them in. When Cucumber tries to execute a step, it looks for a matching step definition to execute.

Keywords are not taken into account when looking for a step definition. This means you cannot have a `Given`, `When`, `Then`, `And` or `But` step with the same text as another step.

Cucumber considers the following steps duplicates:

```
Given there is money in my account
Then there is money in my account
```

This might seem like a limitation, but it forces you to come up with a less ambiguous, more clear domain language:

```
Given my account has a balance of £430
Then my account should have a balance of £430
```

Given

`Given` steps are used to describe the initial context of the system - the *scene* of the scenario. It is typically something that happened in the *past*.

When Cucumber executes a `Given` step, it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to a test database.

The purpose of `Given` steps is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the `When` steps). Avoid talking about user interaction in `Given`'s. If you were creating use cases, `Given`'s would be your preconditions.

It's okay to have several `Given` steps (use `And` or `But` for number 2 and upwards to make it more readable).

Examples:

- Mickey and Minnie have started a game
- I am logged in
- Joe has a balance of £42

When

`When` steps are used to describe an event, or an *action*. This can be a person interacting with the system, or it can be an event triggered by another system.

Examples:

- Guess a word
- Invite a friend
- Withdraw money

> **IMAGINE IT'S 1922**
>
> Most software does something people could do manually (just not as efficiently).
>
> Try hard to come up with examples that don't make any assumptions about technology or user interface. Imagine it's 1922, when there were no computers.
>
> Implementation details should be hidden in the step definitions.

Then

`Then` steps are used to describe an *expected* outcome, or result.

The step definition of a `Then` step should use an *assertion* to compare the *actual* outcome (what the system actually does) to the *expected* outcome (what the step says the system is supposed to do).

An outcome *should* be on an **observable** output. That is, something that comes *out* of the system (report, user interface, message), and not a behaviour deeply buried inside the system (like a record in a database).

Examples:

- See that the guessed word was wrong

- Receive an invitation
- Card should be swallowed

While it might be tempting to implement `Then` steps to look in the database - resist that temptation!

You should only verify an outcome that is observable for the user (or external system), and changes to a database are usually not.

And, But

If you have successive `Given`'s or `Then`'s, you *could* write:

```
Example: Multiple Givens
  Given one thing
  Given another thing
  Given yet another thing
  When I open my eyes
  Then I should see something
  Then I shouldn't see something else
```

Or, you could make the example more fluidly structured by replacing the successive `Given`'s or `Then`'s with `And`'s and `But`'s:

```
Example: Multiple Givens
  Given one thing
  And another thing
  And yet another thing
  When I open my eyes
  Then I should see something
  But I shouldn't see something else
```

*

Gherkin also supports using an asterisk (`*`) in place of any of the normal step keywords. This can be helpful when you have some steps that are effectively a *list of things*, so you can express it more like bullet points where otherwise the natural language of `And` etc might not read so elegantly.

For example:

```
Scenario: All done
  Given I am out shopping
  And I have eggs
  And I have milk
  And I have butter
  When I check my list
  Then I don't need anything
```

Could be expressed as:

```
Scenario: All done
  Given I am out shopping
  * I have eggs
  * I have milk
  * I have butter
  When I check my list
  Then I don't need anything
```

Background

Occasionally you'll find yourself repeating the same `Given` steps in all of the scenarios in a `Feature`.

Since it is repeated in every scenario, this is an indication that those steps are not *essential* to describe the scenarios; they are *incidental details*. You can literally move such `Given` steps to the background, by grouping them under a `Background` section.

A `Background` allows you to add some context to the scenarios that follow it. It can contain one or more `Given` steps, which are run before *each* scenario, but after any Before hooks.

A `Background` is placed before the first `Scenario`/`Example`, at the same level of indentation.

For example:

```
Feature: Multiple site support
  Only blog owners can post to a blog, except administrators,
  who can post to all blogs.
```

```
  Background:
    Given a global administrator named "Greg"
    And a blog named "Greg's anti-tax rants"
    And a customer named "Dr. Bill"
    And a blog named "Expensive Therapy" owned by "Dr. Bill"

  Scenario: Dr. Bill posts to his own blog
    Given I am logged in as Dr. Bill
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

  Scenario: Dr. Bill tries to post to somebody else's blog, and
fails
    Given I am logged in as Dr. Bill
    When I try to post to "Greg's anti-tax rants"
    Then I should see "Hey! That's not your blog!"

  Scenario: Greg posts to a client's blog
    Given I am logged in as Greg
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."
```

`Background` is also supported at the `Rule` level, for example:

```
Feature: Overdue tasks
  Let users know when tasks are overdue, even when using other
  features of the app

  Rule: Users are notified about overdue tasks on first use of the
day
    Background:
      Given I have overdue tasks

    Example: First use of the day
      Given I last used the app yesterday
      When I use the app
      Then I am notified about overdue tasks

    Example: Already used today
      Given I last used the app earlier today
      When I use the app
      Then I am not notified about overdue tasks
```

```
...
```

You can only have one set of `Background` steps per `Feature` or `Rule`. If you need different `Background` steps for different scenarios, consider breaking up your set of scenarios into more `Rule`s or more `Feature`s.

For a less explicit alternative to `Background`, check out conditional hooks.

Tips for using Background

- Don't use `Background` to set up **complicated states**, unless that state is actually something the client needs to know.
  - For example, if the user and site names don't matter to the client, use a higher-level step such as `Given I am logged in as a site owner`.
- Keep your `Background` section **short**.
  - The client needs to actually remember this stuff when reading the scenarios. If the `Background` is more than 4 lines long, consider moving some of the irrelevant details into higher-level steps.
- Make your `Background` section **vivid**.
  - Use colourful names, and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like `"User A"`, `"User B"`, `"Site 1"`, and so on.
- Keep your scenarios **short**, and don't have too many.
  - If the `Background` section has scrolled off the screen, the reader no longer has a full overview of what's happening. Think about using higher-level steps, or splitting the `*.feature` file.

Scenario Outline

The `Scenario Outline` keyword can be used to run the same `Scenario` multiple times, with different combinations of values.

The keyword `Scenario Template` is a synonym of the keyword `Scenario Outline`.

Copying and pasting scenarios to use different values quickly becomes tedious and repetitive:

```
Scenario: eat 5 out of 12
  Given there are 12 cucumbers
```

```
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

Scenario: eat 5 out of 20
  Given there are 20 cucumbers
  When I eat 5 cucumbers
  Then I should have 15 cucumbers
```

We can collapse these two similar scenarios into a `Scenario Outline`.

Scenario outlines allow us to more concisely express these scenarios through the use of a template with `< >`-delimited parameters:

```
Scenario Outline: eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers

  Examples:
    | start | eat | left |
    |    12 |   5 |    7 |
    |    20 |   5 |   15 |
```

Examples

A `Scenario Outline` must contain one or more `Examples` (or `Scenarios`) section(s). Its steps are interpreted as a template which is never directly run. Instead, the `Scenario Outline` is run *once for each row* in the `Examples` section beneath it (not counting the first header row).

The steps can use `<>` delimited *parameters* that reference headers in the examples table. Cucumber will replace these parameters with values from the table *before* it tries to match the step against a step definition.

You can use parameters in `Scenario Outline` descriptions as well.

You can also use parameters in multiline step arguments.

Step Arguments

In some cases you might want to pass more data to a step than fits on a single line. For this purpose Gherkin has `Doc Strings` and `Data Tables`.

Doc Strings

`Doc Strings` are 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:

```
Given a blog post named "Random" with Markdown body
  """
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor
sit amet,
  consectetur adipiscing elit.
  """
```

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.

Doc strings also support using three backticks as the delimiter:

```
Given a blog post named "Random" with Markdown body
  ```
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor
sit amet,
  consectetur adipiscing elit.
  ```
```

This might be familiar for those used to writing with Markdown.

> **TOOL SUPPORT FOR BACKTICKS**
>
> Whilst all current versions of Cucumber support backticks as the delimiter, many tools like text editors don't (yet).

It's possible to annotate the DocString with the type of content it contains. You specify the content type after the triple quote, as follows:

```
Given a blog post named "Random" with Markdown body
  """markdown
  Some Title, Eh?
  ===============
  Here is the first paragraph of my blog post. Lorem ipsum dolor
sit amet,
  consectetur adipiscing elit.
  """
```

> **TOOL SUPPORT FOR CONTENT TYPES**
>
> Whilst all current versions of Cucumber support content types as the delimiter, many tools like text editors don't (yet).

Data Tables

`Data Tables` are handy for passing a list of values to a step definition:

```
Given the following users exist:
  | name   | email              | twitter         |
  | Aslak  | aslak@cucumber.io  | @aslak_hellesoy |
  | Julien | julien@cucumber.io | @jbpros         |
  | Matt   | matt@cucumber.io   | @mattwynne      |
```

Just like `Doc Strings`, `Data Tables` will be passed to the step definition as the last argument.

Table Cell Escaping

If you want to use a newline character in a table cell, you can write this as `\n`. If you need a `|` as part of the cell, you can escape it as `\|`. And finally, if you need a `\`, you can escape that with `\\`.

Data Table API

Cucumber provides a rich API for manipulating tables from within step definitions. See the Data Table API reference reference for more details.

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 .

Here is a Gherkin scenario written in Norwegian:

```
# language: no
Funksjonalitet: Gjett et ord

  Eksempel: Ordmaker starter et spill
    Når Ordmaker starter et spill
    Så må Ordmaker vente på at Gjetter blir med

  Eksempel: Gjetter blir med
    Gitt at Ordmaker har startet et spill med ordet "bløtt"
    Når Gjetter blir med på Ordmakers spill
    Så må Gjetter gjette et ord på 5 bokstaver
```

A `# language:` header on the first line of a feature file tells Cucumber what spoken language to use - for example `# language: fr` for French. If you omit this header, Cucumber will default to English (`en`).

Some Cucumber implementations also let you set the default language in the configuration, so you don't need to place the `# language` header in every file.

Edit this page

*Last updated on **Jan 26, 2025***