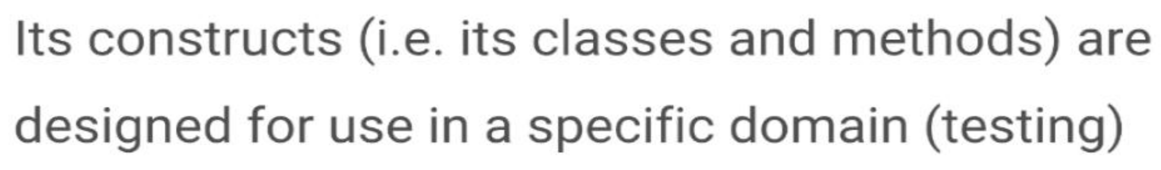
Section 1

1. Welcome to RSpec

RSpec -> ruby testing library. The most popular ruby gem.

It is a DSL

A DSL is a language with a high level of abstraction built to solve a really specific problem.
In this case, the problem is testing.



Its constructs (i.e. its classes and methods) are designed for use in a specific domain (testing)

Why to test code?

Testing here is to create ruby code that will check other ruby code and assure it behaves as expected.

Testing by hand means a lot of work, not finding most of the bugs, etc.

- Automatic testing helps to avoid regression when adding new features.
- Also provides documentation. Avoid writing documentation since developers can see what a class does by watching the test code.
- Helps to localizate problems.

TDD

Test driven development
It is building your test, before making your code.

It forces you to think about how you are going to write your code. How many methods, which methods a class will have.

You write the feature, later the test, the test is going to fail because there is no code, then you build the code, and then it is going to pass. Repeat these steps and you are going to create a solid product.

It will make you a better developer
Resource about TDD
https://www.madetech.com/blog/9-benefits-of-test-driven-development/

The RSpec ecosystem

Consist of multiple gems that can be used alone.

- rspec-core
    - Foundation, it runs the test. How many tests are failing. What is the slower test, etc.
- rspec-expectations
    - Provides syntax to create your assertions. We write how we expect something to work.
- rspec-mocks
    - Utilities to fake behavior of classes.

Why to have mock-ups?
Usually there are a lot of classes that are connected. When we create a unit test we want to test these clases alone and we should create the other pieces by ourselves.

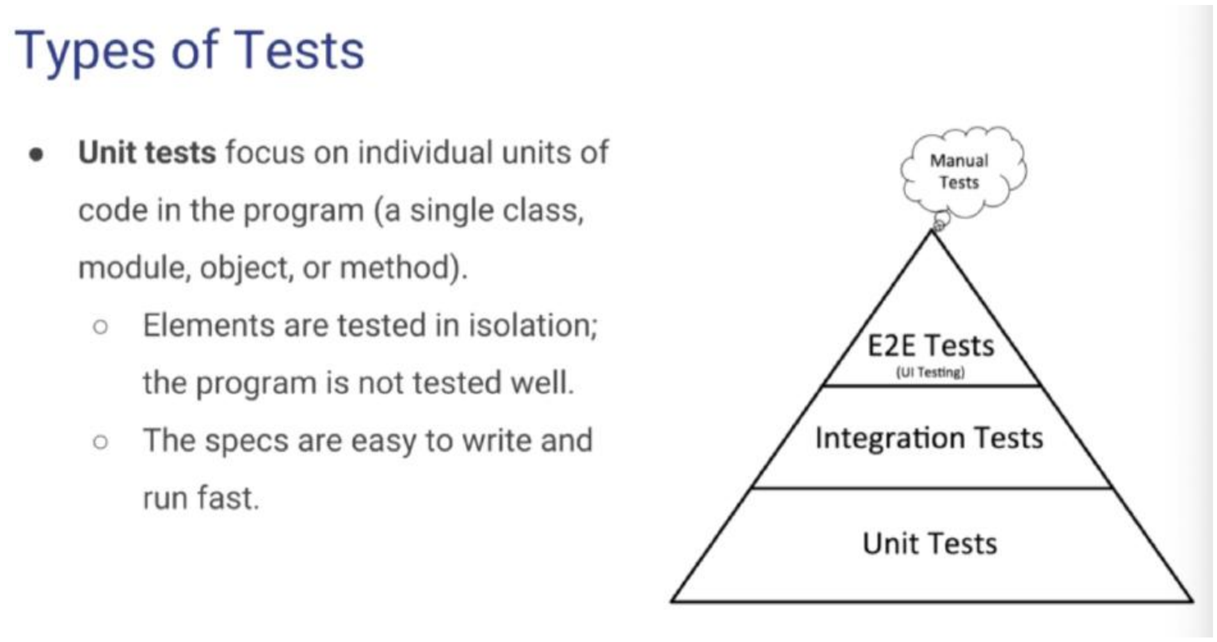
We can fake a call to the database for example.

There is also the **rspec-rails** that integrates Rspec with the ruby on rails web framework.

Project structure

- spec directory
    - House all rspec test files
- Nested directories inside spec
    - Mimic the lib folder (ruby projects) or app folder (rails projects)
    - If there is a app/model/knight.rb file, in this directory there will be a app/model/knight_spec.rb file.

3. Unit Tests vs End-to-End (E2E) Tests



Types of Tests

- **Unit tests** focus on individual units of code in the program (a single class, module, object, or method).
  - Elements are tested in isolation; the program is not tested well.
  - The specs are easy to write and run fast.

The image reflects the importance of each test.
Unit tests should be the bigger percentage of tests.

Unit test

Focus on a specific unit of code.
A class, model, or even a method.
Goal is to test a piece of the code in total isolation.

We cannot assure the stability of the whole system, if one of the pieces is broken by itself.
We should make sure that each feature works by itself.
These tests are easy to do and fast to execute.

E2E test

Test a feature of a program from the start, to the end.
A E2E test can be:

- Login
- Putting items into a shopping car.
- Enter your credit card and confirm that the page shows a successful message.

It is an integrated test.

Advantage:
    You test something similar to what the user will do.
Disadvantage
    If something breaks, you do not know where it broke.

Tests are harder to create.

Unit test is testing the wheel of a car while the E2E is testing the whole car.

Integration tests

Falls in the middle of these two.
You are not testing something by itself, but neither testing the whole programm.
Integration is like a feature and integration is like the entire program or an entire process flow.

Manual test

There can still be some manual tests. It is at the top of the image because we should focus on the computer to do the tests.

On a project there are money or time limitations, but we should try to do all types of tests if it is possible.

Setup Ruby on macOS Computers

The recommended way to setup ruby is with rbenv.
To install it on mac, use homebrew.
Homebrew is a piece of software that simplifies the process of installing and configuring other pieces of software for your computer.

…

6. Install RSpec

You need rbenv already installed.

Gem
Ruby package manager.
Fetch's ruby gems and libraries for us.

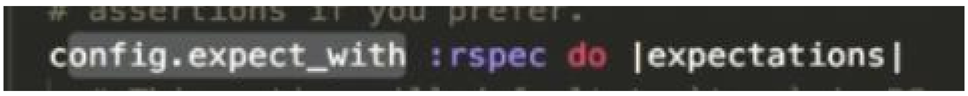
gem install rspec

8. Start a Project with rspec --init

rspec --init
Creates some files

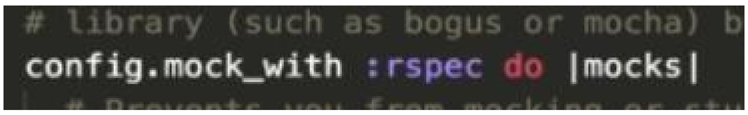
- .rspec
    - Only include only one line. The line corresponds with the spec_helper on the other directory.
    - It ensures that whatever we run rspec on the command line. It ensures to run spec_helper first before any test case.
    - It also means that you do not have to require or import spec_helper.
- spec/spec_helper.rb
    - A lot of configurations.
    - Only add global things here since it will run on every type of test. A large file can make tests slow.
    - On that file we can customize what gems we want in case we want to replace the other rspec gems.

This line tells rspect to use his default spectations (asserts) gem.



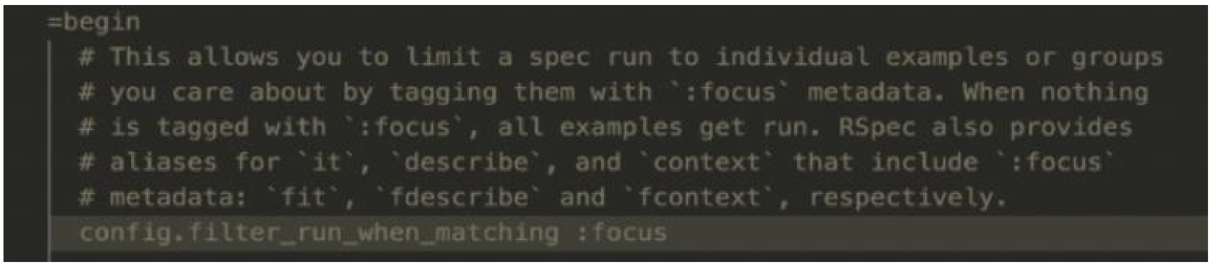
```
  # assertions if you prefer.
  config.expect_with :rspec do |expectations|
```

This line tells rspect to use his default mockup gem



```
# library (such as bogus or mocha) b
config.mock_with :rspec do |mocks|
```

There is commented code with the recommended configuration in case you want to use it



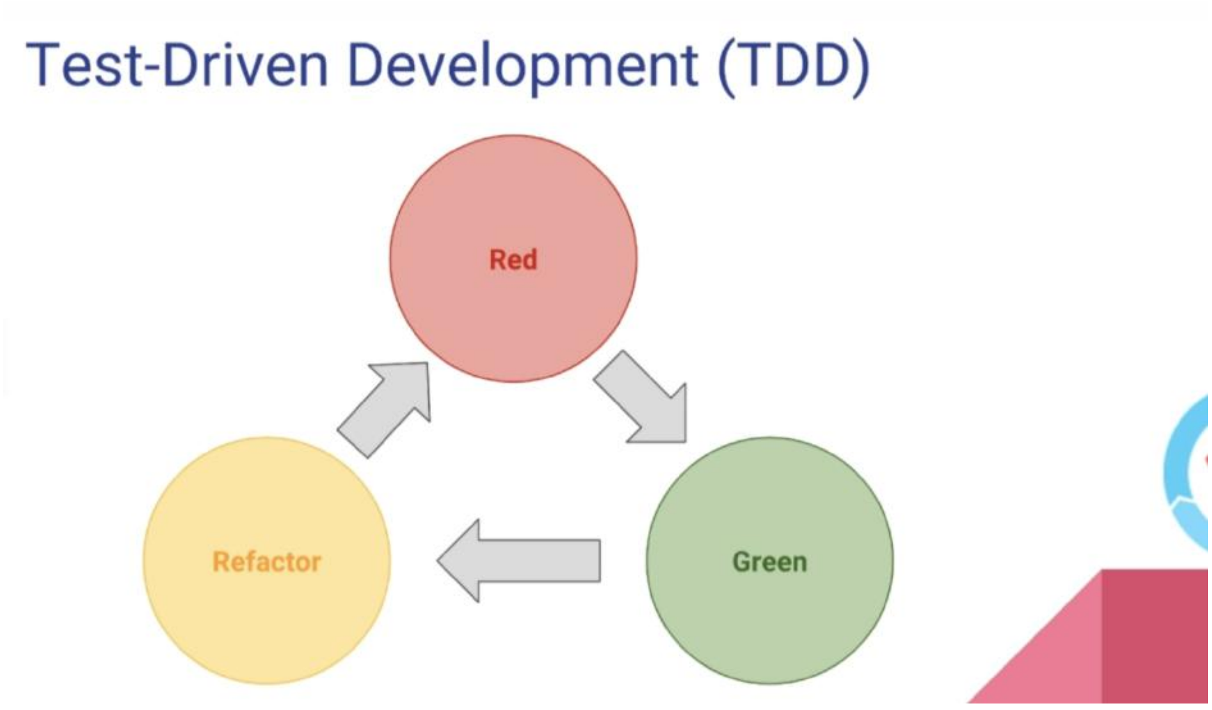
```
=begin
  # This allows you to limit a spec run to individual examples or groups
  # you care about by tagging them with `:focus` metadata. When nothing
  # is tagged with `:focus`, all examples get run. RSpec also provides
  # aliases for `it`, `describe`, and `context` that include `:focus`
  # metadata: `fit`, `fdescribe` and `fcontext`, respectively.
  config.filter_run_when_matching :focus
```

About the class

In the real world we put the original files and the tests file on separate directories.
In this course we will have the code mixed to simplify things. Everything in one file.

10. Test-Driven Development

TDD says that you should write your tests first.



The goal of TDD is to **force you to think how something should work and create it incrementally, piece by piece, chunk by chunk, method by method**.

We usually want to rush and start adding code. And we dont worry about class or method responsibility. TDD helps you to slow you down.

we begin on red (fail), we write code and it fails because there is no code. We write the bare minimum code to pass the test (green / success). After that we refactor so we do it cleaner.

Practicing this, help you to become a better developer.

TDD also ensures that everything is covered by tests.

Makes you think like: Ok, I need to test this, I need a method, what parameters this method requires, what order of execution it will have, etc. **Helps you to structure your programm by small pieces of code.**

11. The described Method

SPect module

Normally you have to write first the specs (the card_spec file instead of the card file).

<u>If we run the spec file from the command line, we will have access to our spec as a globally defined module.</u>
We have “RSpec” available to us.

Describe method

It is from everything that kicks off. It describes what we are testing.

```
RSpec.describe 'Card'
```

In this case we are testing the card class.

Example group

It accepts a block.
It creates an **example group**
Contains a related collection of tests.

A test is also called an example (it is an example of how something should work).
So an example group is a group of tests. **Related tests**

In other words, we are saying that we are going to describe the “card” functionality inside the code block.

12. The it Method

It -> Allows us to create a specific test. Specific **functionality**.

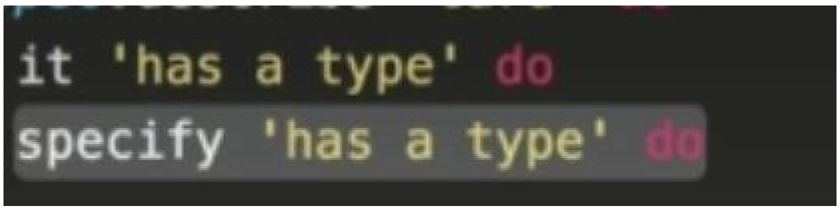
Called like that because it receives a string and you should combine that string with the word “it”.
You describe the expected behavior.
we talk about what it is doing instead of how it is going to be made (it has an array, etc).

Where we write the implementation is where we write the expectations on the body of its method.

In the real world sadly you will sometimes have to write a long description, that is fine.

Other way of writing the same

They are both identical.



```
it 'has a type' do
specify 'has a type' do
```

13. The expect and eq Methods

We think ahead of development and suppose what this class will have. This time we a guessing that we pass the type to the constructor method and that the type is a string:

```ruby
card = Card.new('Ace of Spades')
```

We assume also that there is also a public global attribute called type.

Expect method

What are we validating?.

Below we are saying “I expect the card type to be equals to “Ace of Spades””

```ruby
expect(card.type).to eq('Ace of Spades')
```

The point is that the language syntax looks like an English phrase.

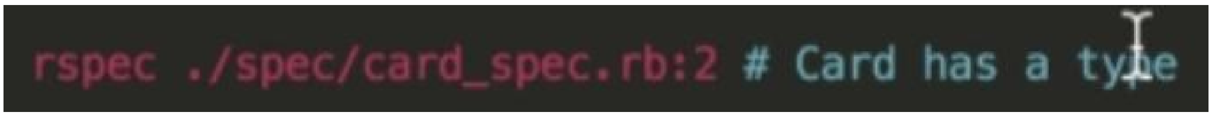
Expect -> what are we gonna evaluate. (It can be method, value, expression)
to -> We invoke this all the time
eq -> checks that the value is equal to something. We pass it to the “to” method.

14. Reading Failures

rspec
It will run every single rspec file inside the spec file.

rspec spec/card_spec.rb
Runs only that file.

Running that command will give you another command on fail:

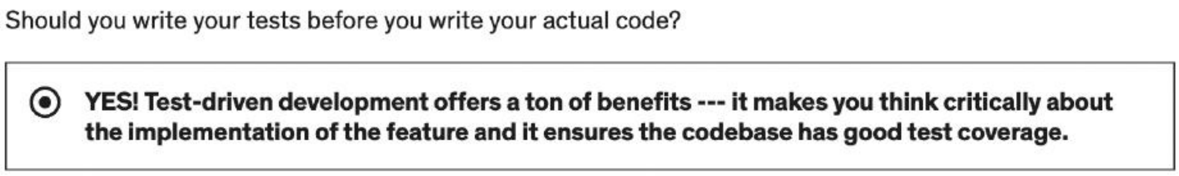


```
rspec ./spec/card_spec.rb:2 # Card has a type
```

This command runs the specific test that failed.

15. Making the Specs Pass

We should note that we did not know how the class was going to look. We first write down what we expect from it.
Using those expectations, we write the code.

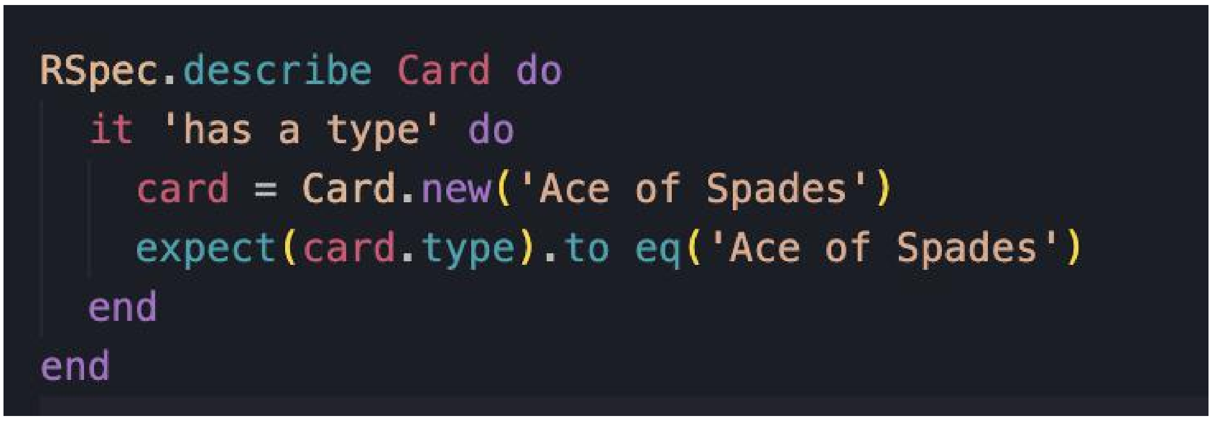


Should you write your tests before you write your actual code?

[✓ **YES! Test-driven development offers a ton of benefits --- it makes you think critically about the implementation of the feature and it ensures the codebase has good test coverage.**]

16. Multiple Examples in Example Group

Instead of giving the “describe” method a string, you can give it a class, array, etc.



```ruby
RSpec.describe Card do
  it 'has a type' do
    card = Card.new('Ace of Spades')
    expect(card.type).to eq('Ace of Spades')
  end
end
```

The advantages of that:
This is going to generate some helper methods that help to reduce the quantity of code.

We are going to see that later, for the moment only stick to your head the idea that it is better to give a class than a string. Whenever it is possible.

One exception per example?

Some people say that having multiple expectations means that you are testing too much on it. It is debatible.
It depends on your team or you to do it that way.
For this case we are going to do it that way, with both examples

Every example has its variables isolated.

17. Fixing Failing Specs Again

Code repetition or not?

Advantages

You do not contaminate an instance of the object you are testing with code repetition since each test probably is going to modify it.

Disadvantages

If you have 20 examples, you will have to write a lot of the same code.

What to do?

Next chapter we are going to see what we can do.

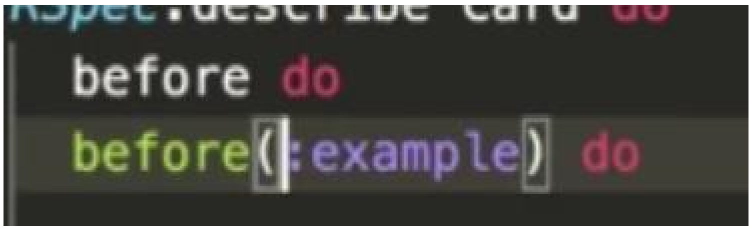
18. Reducing Duplication - Before Hooks and Instance Variables

A way to reduce duplication

This way is not the best one, the optimal one.
But probably we are going to find it on other people's codes.

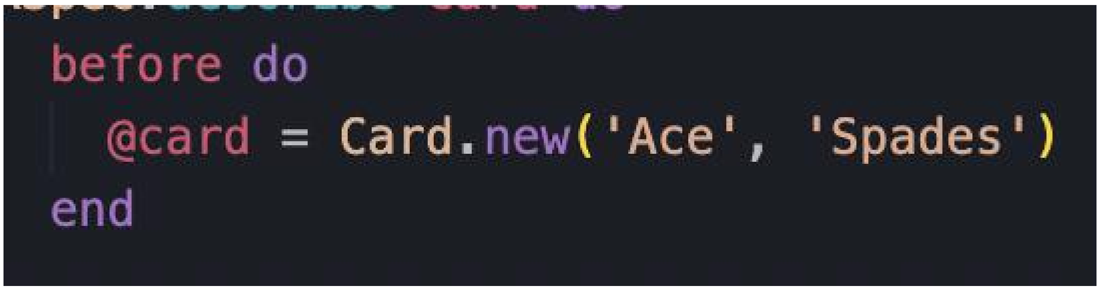
Before Hooks

Piece of code that runs automatically depending on an event.
Two ways of defining this



```
  before do
  before(:example) do
```

It will run on every test execution.
This before instruction will help us to write some setup code.



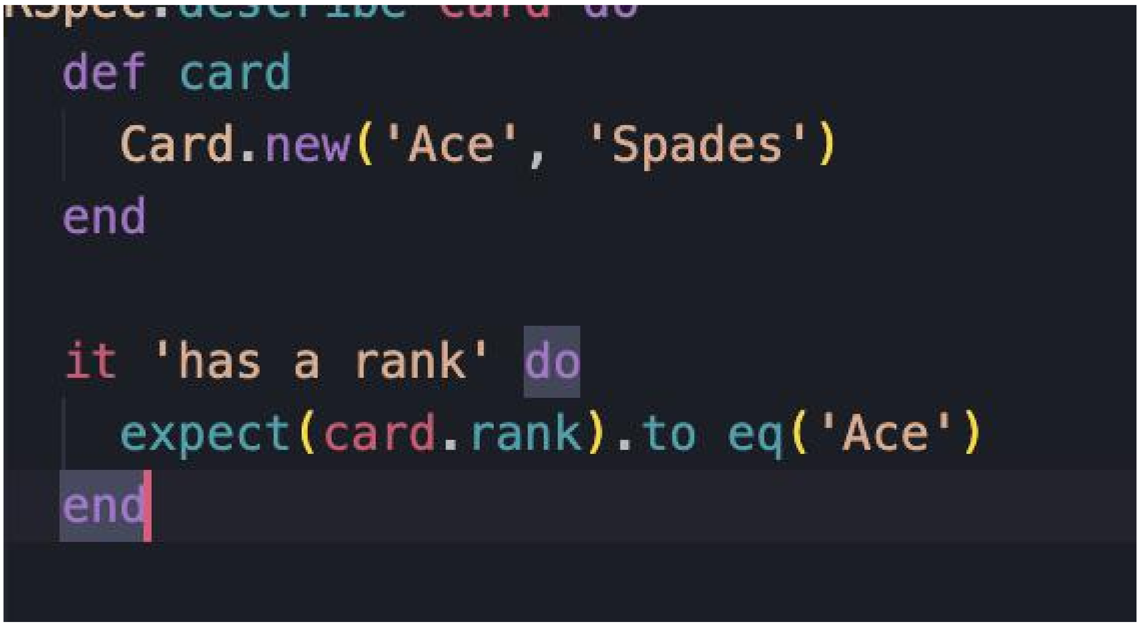
```
  before do
    @card = Card.new('Ace', 'Spades')
  end
```

Why is this not the preferable way to reduce repetition?

If by error, @card does not exist. Then it will not throw an error, it will be nil, and you are going to end up on a validation with nil. Things that can end up passing the test automatically depending on the validation.

19. Reducing Duplication: Helper Methods

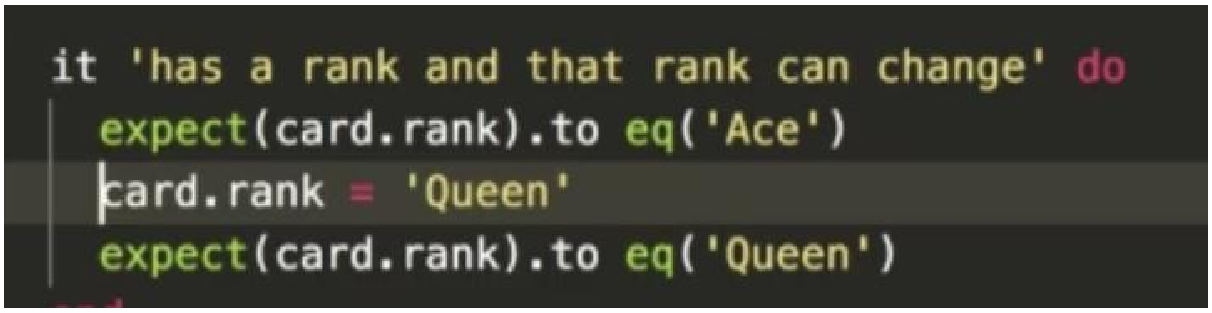
Create a method that returns the instance to fix the problem.
If you write it badly, then it is going to throw an error.



```ruby
  def card
    Card.new('Ace', 'Spades')
  end

  it 'has a rank' do
    expect(card.rank).to eq('Ace')
  end
```

20. Problems With Mutation

Every time we use “card” ends up being another instance



```ruby
  it 'has a rank and that rank can change' do
    expect(card.rank).to eq('Ace')
    card.rank = 'Queen'
    expect(card.rank).to eq('Queen')
```

21. Reducing Duplication: The let Method

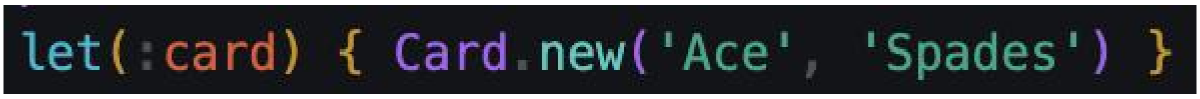
Memorization

It is like a cache, if the computer instances an object, it will not instance it again, it is going to use the same one.

If you have a math problem and you solve it. Then the teacher gives you the same problem. Will you resolve it again?

Computers do the same.

We can achieve this with the method “let”. It is a method from RSpec



```ruby
let(:card) { Card.new('Ace', 'Spades') }
```

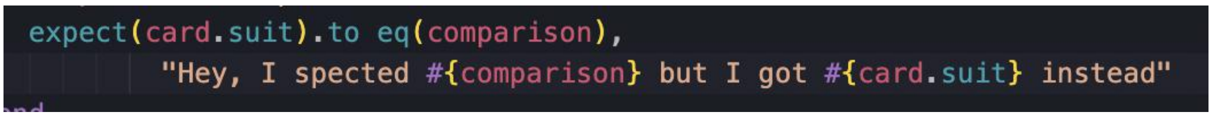
We pass a symbol and we can think of the symbol like a variable that is going to be created. The block on the right is to assign the value.

1. Between each example, this code is going to be run and instance a new card.
2. Now the card on each example is going to use the same object.
3. Also we should mention is that we are using lazy running
    a. This line is not going to run when the program starts, it will run on the first example the first time.
    b. So this variable is going to be created only when needed
    c. If we have 100 examples without that variable, it is ok since it will not run.

Aun así si quieres crearla antes del ejemplo
Agrega un bang al let (let!)
Esto lo hace comportarse como un <u>before</u>, habrá ocasiones que lo queramos así.

22. Custom Error Messages

For business rules or to explain better an assertion error, pass another parameter to the “to” method.



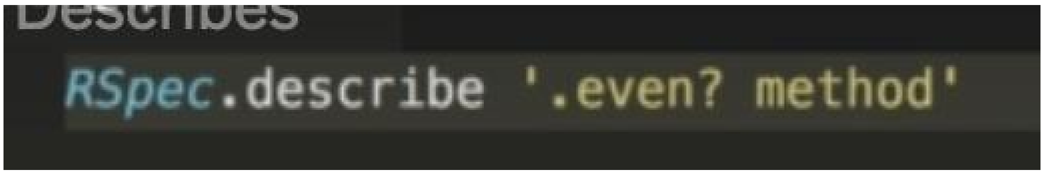
```
expect(card.suit).to eq(comparison),
            "Hey, I spected #{comparison} but I got #{card.suit} instead"
```

23. The context Method and Nested Describes

As community standar:

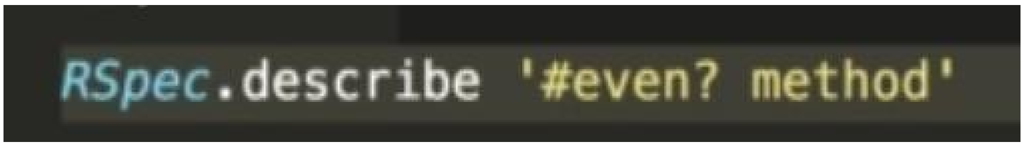
Testing a class method

Use a period before the name



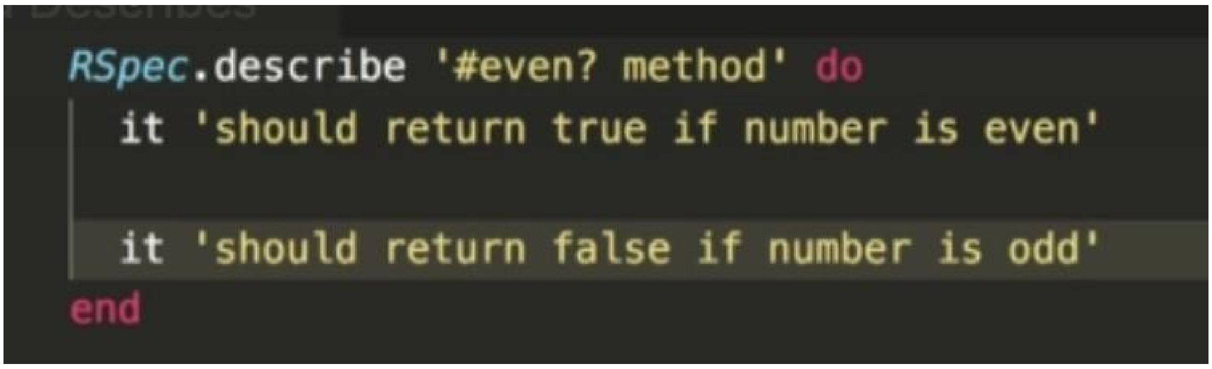
```
RSpec.describe '.even? method'
```

Instance method



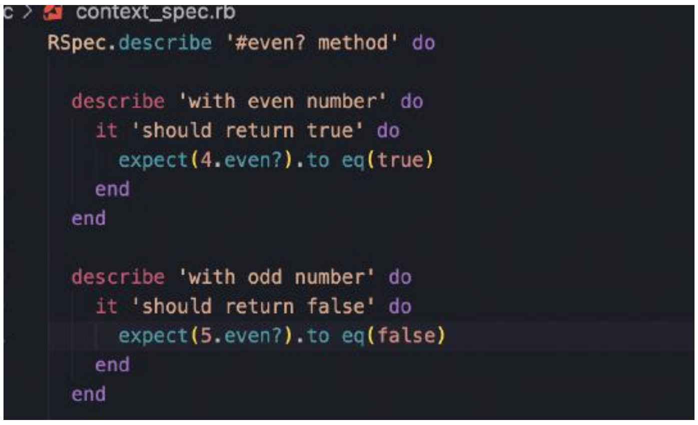
```
RSpec.describe '#even? method'
```

You can write the examples like this:
This is fine. But not the best way to organize it, it has conditions and the person reading the text should know that not only should return true, but also return true if even. It adds complexity.



```
RSpec.describe '#even? method' do
  it 'should return true if number is even'

  it 'should return false if number is odd'
end
```

A way to fix it is to write a nested block
You can write describe inside to organize the project better.



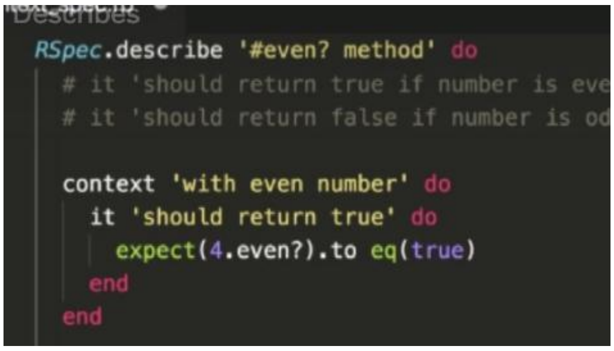
```
c > context_spec.rb
RSpec.describe '#even? method' do

  describe 'with even number' do
    it 'should return true' do
      expect(4.even?).to eq(true)
    end
  end

  describe 'with odd number' do
    it 'should return false' do
      expect(5.even?).to eq(false)
    end
  end
```

It is easier to read.

If you use 'if', 'when' or 'this condition', it is a sign you should abstract this to a higher level description block.

Context

It is the same as describe, but a lot of developers used it on nested descriptions because it is more semantic correct.



```
Describes
RSpec.describe '#even? method' do
  # it 'should return true if number is eve
  # it 'should return false if number is od

  context 'with even number' do
    it 'should return true' do
      expect(4.even?).to eq(true)
    end
  end
```

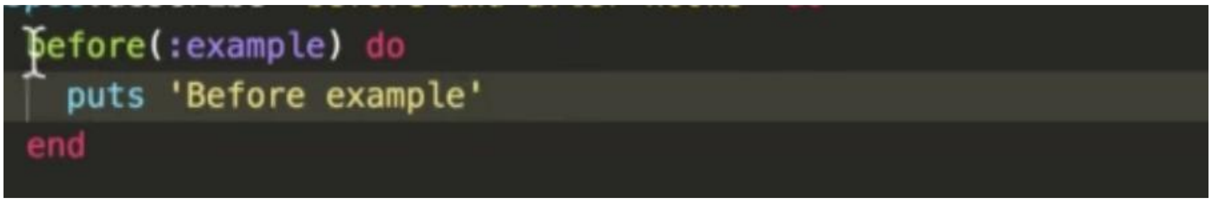
You can use as many contexts of description as you think it will help to make the code easier to read.

24. before and after Hooks

We are going to explore more about hooks
A hook allows us to execute a block of code at certain time on the execution

Before

Will run before any example



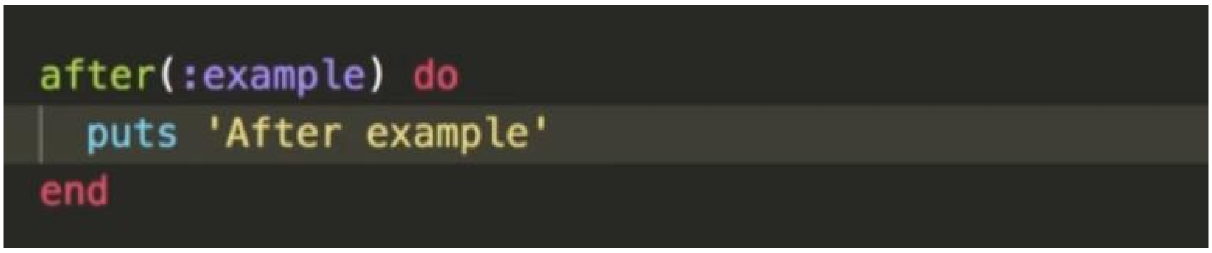
```ruby
before(:example) do
  puts 'Before example'
end
```

After

Similar to before, it runs after each example

Useful since some teams prioritize speed, and they do not want to be instantiating a new object on each example, specially it is a heavy object, so they reset it after each example.

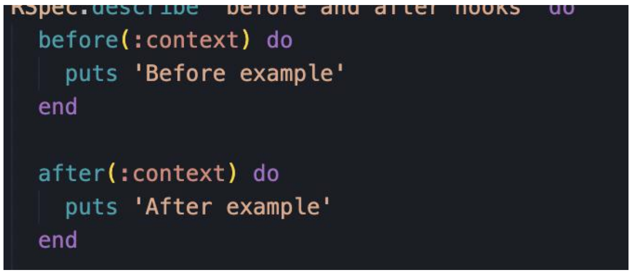
He does not really prefer this way, he prefers a new clean object on each example, but it depends on how your team do the things.



```ruby
after(:example) do
  puts 'After example'
end
```

Before and after context

IT only runs one on the context it is in.
With context you can think of it as the current block it is in.



```ruby
RSpec.describe 'before and after hooks' do
  before(:context) do
    puts 'Before example'
  end

  after(:context) do
    puts 'After example'
  end
```

With after context maybe you want to print a message or close a database connection for example.

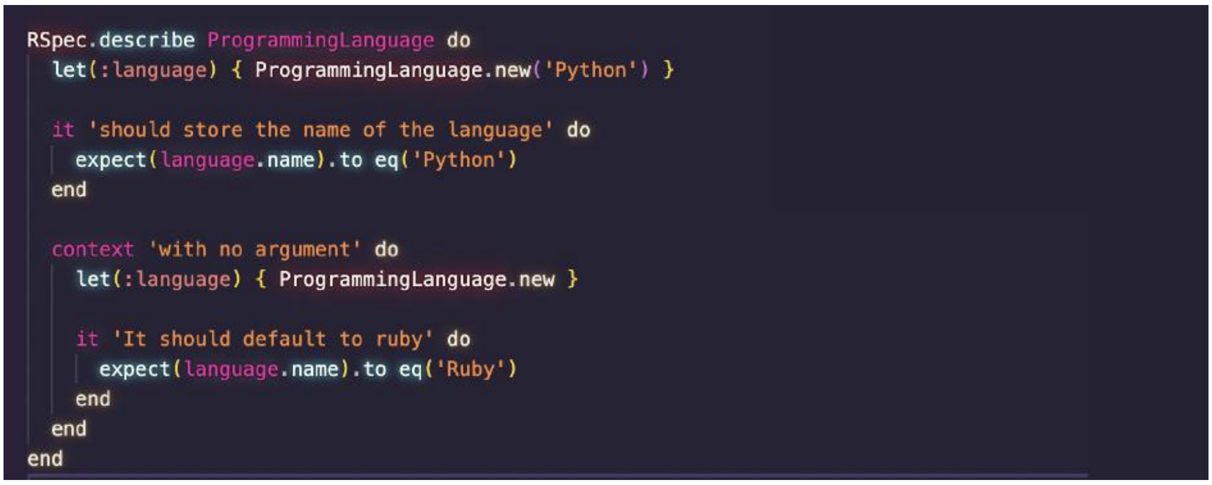
25. Nested Logic: Hooks

Important

All outer levels before and after will run also on inner tests. This only happens with ‘:example’ and not with :context.

26. Nested Logic: Overwriting Let Variables

You can overwrite a let variable on another block, by writing again the let method on that block.



```ruby
RSpec.describe ProgrammingLanguage do
  let(:language) { ProgrammingLanguage.new('Python') }

  it 'should store the name of the language' do
    expect(language.name).to eq('Python')
  end

  context 'with no argument' do
    let(:language) { ProgrammingLanguage.new }

    it 'It should default to ruby' do
      expect(language.name).to eq('Ruby')
    end
  end
end
```