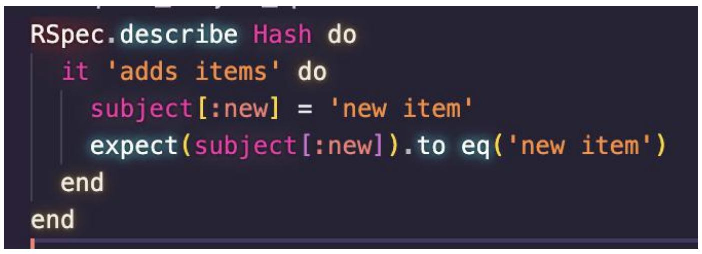
27. Implicit Subject

Session focused on reducing boilerplate code from our specs.
He means reducing duplication.

When we pass the class name to the describe method, it creates an instance of that class for us.
It behaves exactly as let. The object is created again for each example.
This allows us to reduce code.

Of course there will be times where we have to customize the object, but it is good to know this.

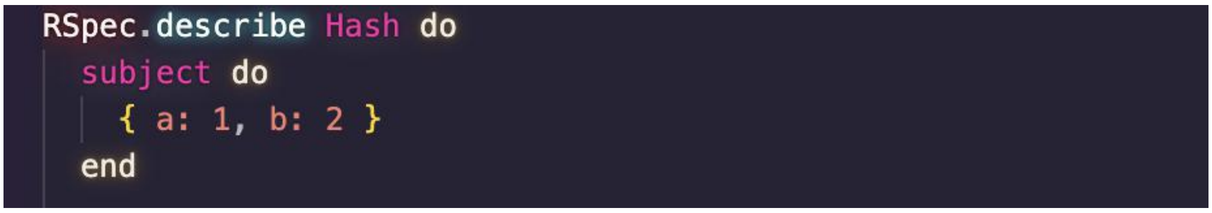
On TDD probably we will not be able to use classes because the class may not exist. There is no problem, use a string.



```ruby
RSpec.describe Hash do
  it 'adds items' do
    subject[:new] = 'new item'
    expect(subject[:new]).to eq('new item')
  end
end
```

28. Explicit Subject

Customize the subject.

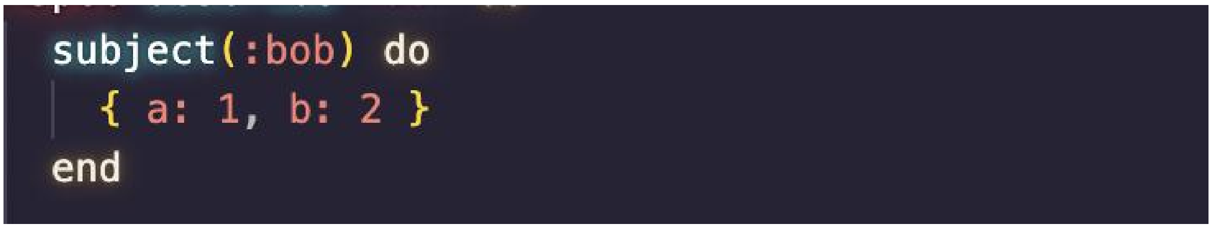


```ruby
RSpec.describe Hash do
  subject do
    { a: 1, b: 2 }
  end
```

You can have more than one subject but he thinks it is more professional to only have one.

Add alias to subject

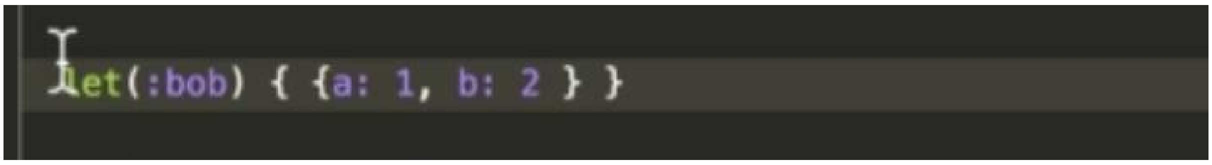
Subject will be still available but you can also use “bob” as on the example:



```ruby
subject(:bob) do
  { a: 1, b: 2 }
end
```

The same as let

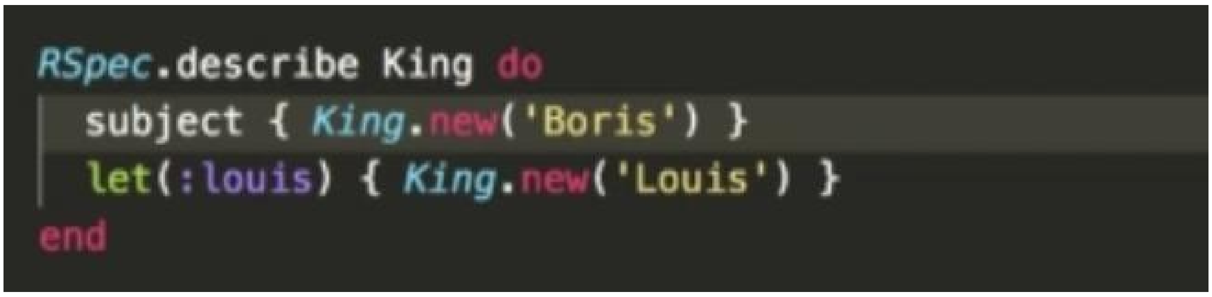
You can create it with let, nobody is going to tell you something but we are going to see in the future some shortcuts that subject give us.



```ruby
let(:bob) { {a: 1, b: 2 } }
```

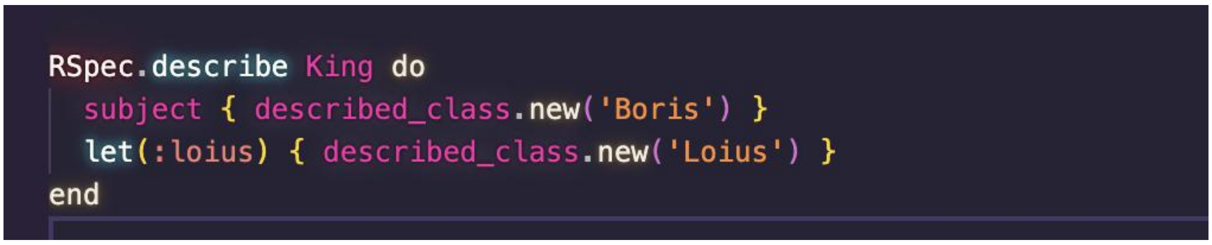
29. described_class

Instead of writing the class name while having the risk of a change of name from the class. We can used described_class
Without described:



```ruby
RSpec.describe King do
  subject { King.new('Boris') }
  let(:louis) { King.new('Louis') }
end
```

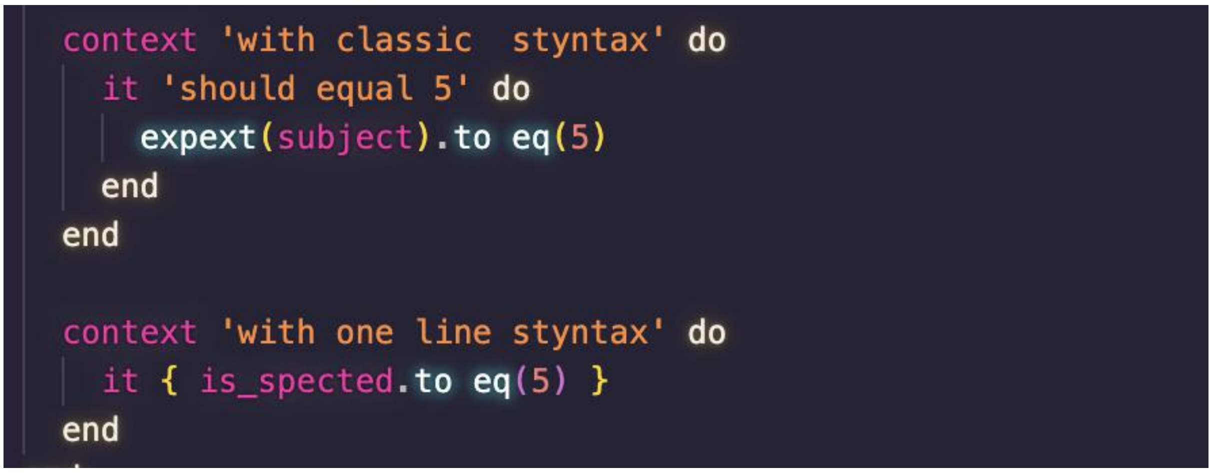
With described:



```ruby
RSpec.describe King do
  subject { described_class.new('Boris') }
  let(:loius) { described_class.new('Loius') }
end
```

30. One-Liner-Example-Syntax

Another advantage of using subject over let
You can omit the message inside the it (an automatic message will appear on run)



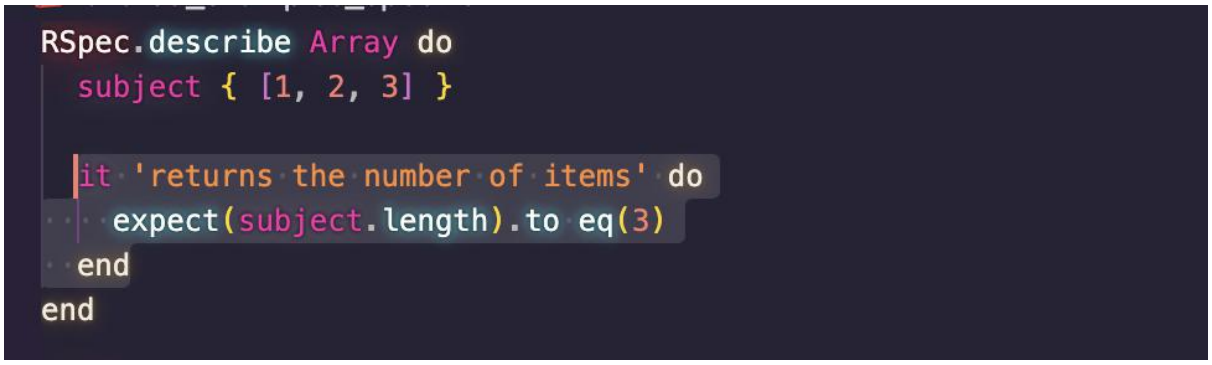
```ruby
context 'with classic  styntax' do
  it 'should equal 5' do
    expext(subject).to eq(5)
  end
end

context 'with one line styntax' do
  it { is_spected.to eq(5) }
end
```

31. Shared Examples with include_examples

Let's suppose we want to test the length method of the array, string and hash, all of them in their respective describe method.

Doing the next it on each describe is repetitive code



```ruby
RSpec.describe Array do
  subject { [1, 2, 3] }

  it 'returns the number of items' do
    expect(subject.length).to eq(3)
  end
end
```

Instead of writing three identical examples, we are going to create a **shared example**

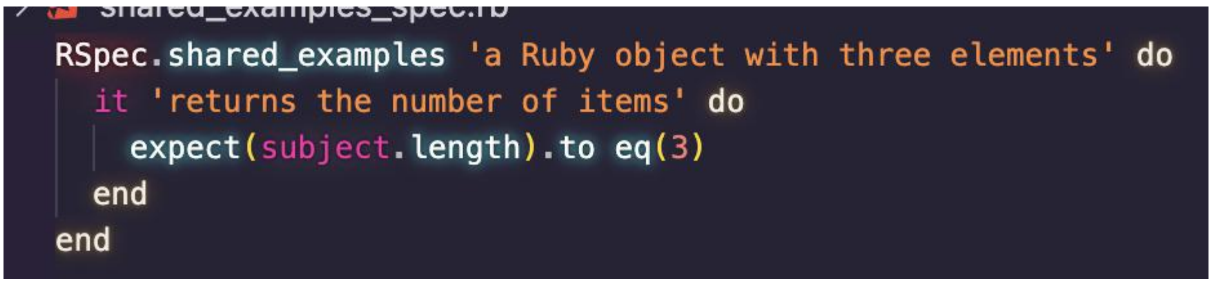
In a real app, we are going to have a file where we are going to declare a lot of shared examples.

To keep the course simple, we are going to add the shared example on the same file.

Create one

We use shared_examples (preferable on another file) and we give it a name, we add the shared example inside the block

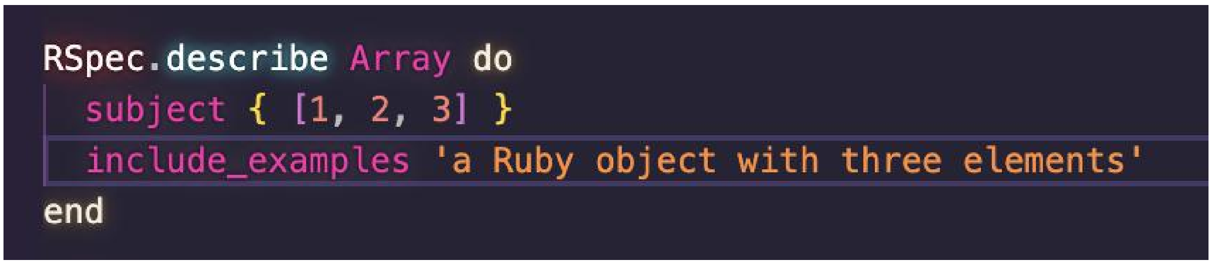
You can add more than one example.



```ruby
RSpec.shared_examples 'a Ruby object with three elements' do
  it 'returns the number of items' do
    expect(subject.length).to eq(3)
  end
end
```

Use the shared example

We use the function include example and we use the name we gave it.



```ruby
RSpec.describe Array do
  subject { [1, 2, 3] }
  include_examples 'a Ruby object with three elements'
end
```

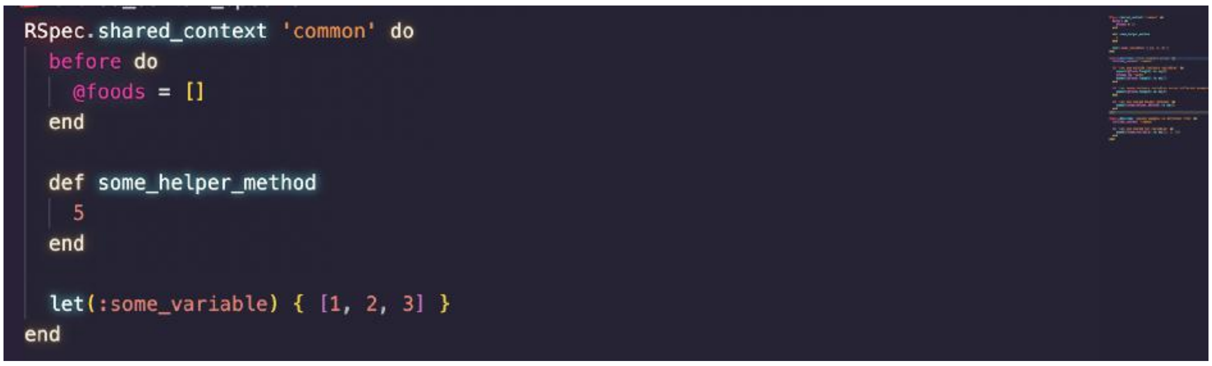
32. Shared Context with include_context

Shared helper methods, let methods, etc. That can be injected into other example groups.
Shared contexts are also added into another file.
Useful to define common setup.

Share:

- Helper methods
- Let method
- hooks

Create

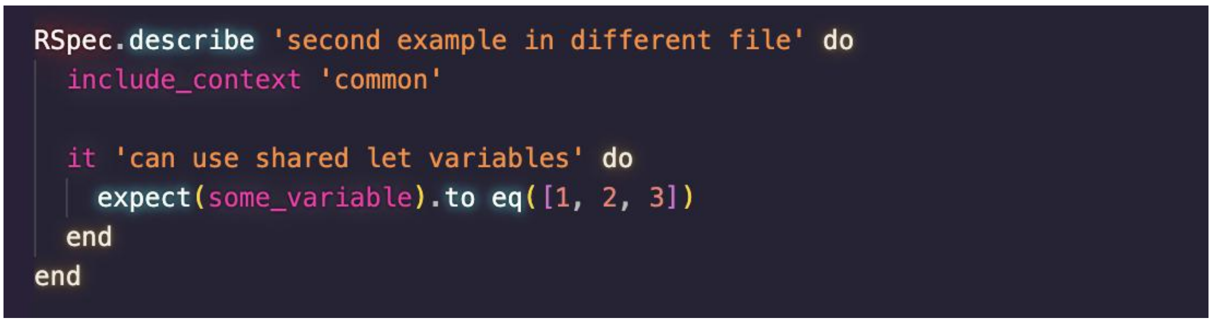


```ruby
RSpec.shared_context 'common' do
  before do
    @foods = []
  end

  def some_helper_method
    5
  end

  let(:some_variable) { [1, 2, 3] }
end
```

Use



```ruby
RSpec.describe 'second example in different file' do
  include_context 'common'

  it 'can use shared let variables' do
    expect(some_variable).to eq([1, 2, 3])
  end
end
```