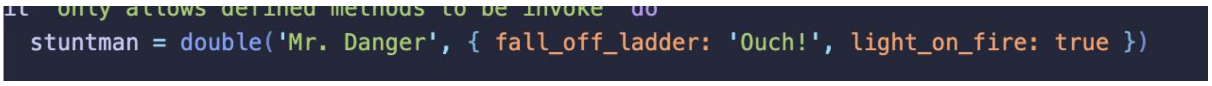
Section 3

This section is about mocking.

50. Create a Test Double

Mocking is about to simulate something.
It is hard to test a unit because a class usually has a lot of dependencies. So we replace the dependencies of the class we are testing for mockups.

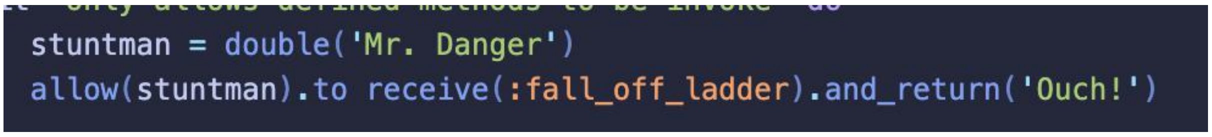
We use double to create mockups, think of the reason for the name as the double from a movie.



```
it 'only allows defined methods to be invoke' do
  stuntman = double('Mr. Danger', { fall_off_ladder: 'Ouch!', light_on_fire: true })
```

You pass the double an identifier and a hash with the name of methods and return value.

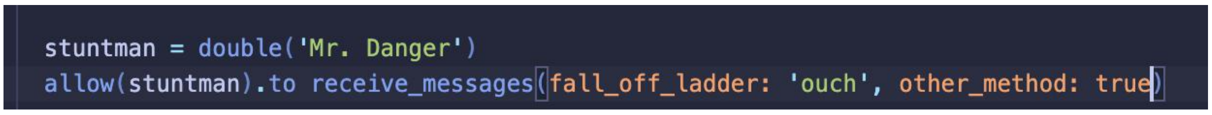
Alter syntax



```
it 'only allows defined methods to be invoke' do
  stuntman = double('Mr. Danger')
  allow(stuntman).to receive(:fall_off_ladder).and_return('Ouch!')
```

Other syntax

Similar to the first one where you pass a hash.



```
  stuntman = double('Mr. Danger')
  allow(stuntman).to receive_messages(fall_off_ladder: 'ouch', other_method: true)
```

51. Set up Our Test Movie

TDD also allows you to write mockups of your other objects in the system.

You usually do not want to make queries to the database or do fetches across the internet since it will decrease the performance and speed of the tests.

If we are going to test a movie and it depends on the actor. We create a double of actors, why?. We do it because we are testing a movie, and we do not really care about what an actor does since we are testing the movie.

The internals of another class is irrelevant to the class we are testing.

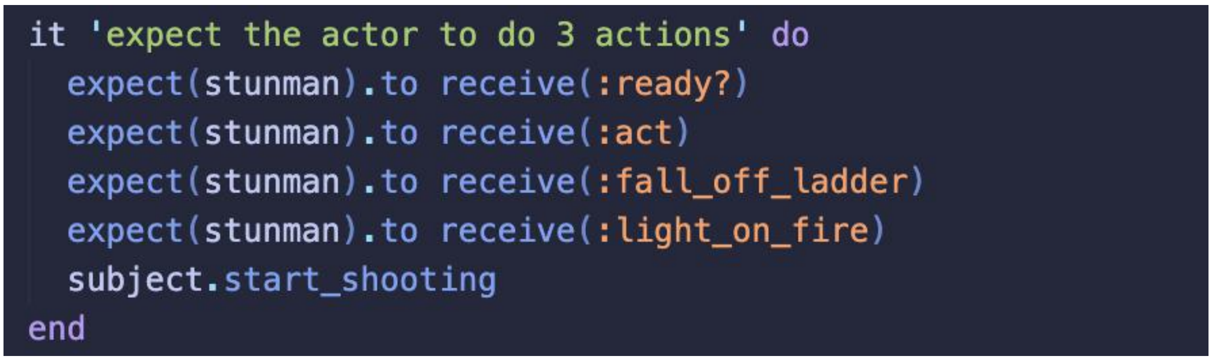
52. Replacing an Object with a Double

On his example he is going to create a double for “Actor” for these reasons:

- If there is an error, it is more difficult to know if the error is actually from the “Movie” class and not the “Actor” class.
- The actor class can be actually very heavy.

Receive method

Makes sure that the method is being called.



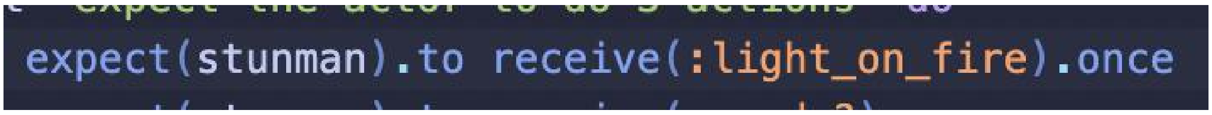
```
it 'expect the actor to do 3 actions' do
  expect(stunman).to receive(:ready?)
  expect(stunman).to receive(:act)
  expect(stunman).to receive(:fall_off_ladder)
  expect(stunman).to receive(:light_on_fire)
  subject.start_shooting
end
```

53. Receive Counts

We want to check that a method call is calling another class method twice.

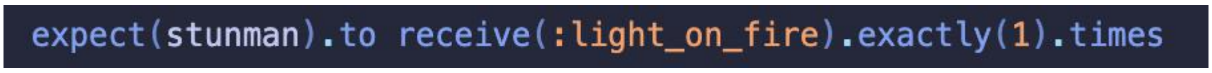
Once

We assure a method is called once, if zero or more than one calls are done, then it fails.



```
expect(stunman).to receive(:light_on_fire).once
```

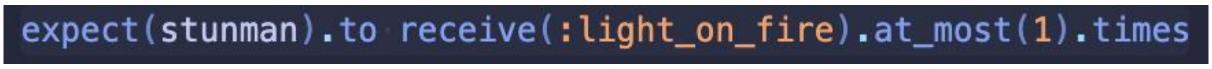
exactly(n).times



```
expect(stunman).to receive(:light_on_fire).exactly(1).times
```

at_most

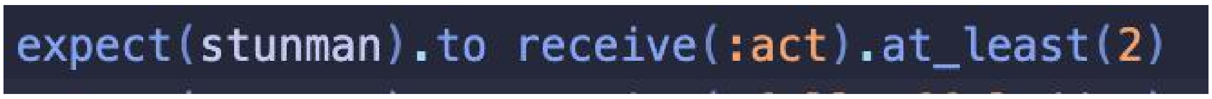
Put a limit to the number of times a method can be called



```
expect(stunman).to receive(:light_on_fire).at_most(1).times
```

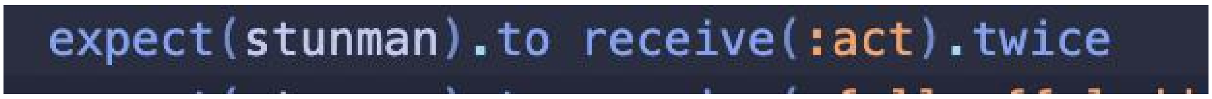
at_least

Setup a minimum quantity of calls.



```ruby
expect(stunman).to receive(:act).at_least(2)
```

twice



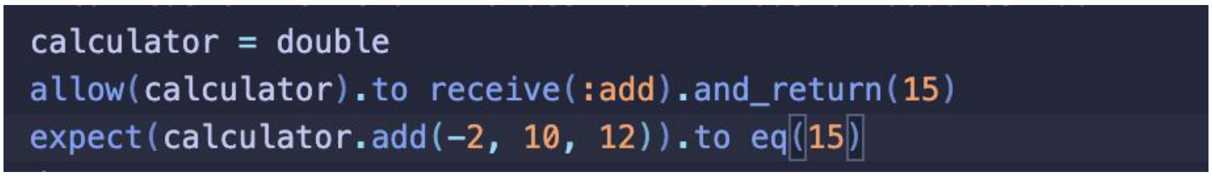
```ruby
expect(stunman).to receive(:act).twice
```

54. The allow Method

This method allows us to pair a method to a given return value.
We can add methods not only on doubles but with real ruby objects also

This is so you can maintain the other methods of a real class, except some of them.

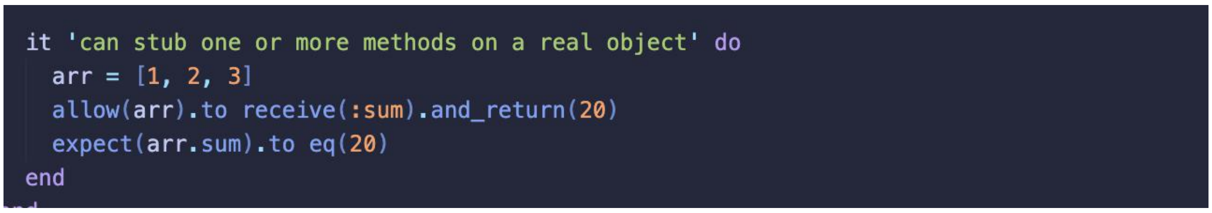
Used with a double



```ruby
calculator = double
allow(calculator).to receive(:add).and_return(15)
expect(calculator.add(-2, 10, 12)).to eq(15)
```

Real object

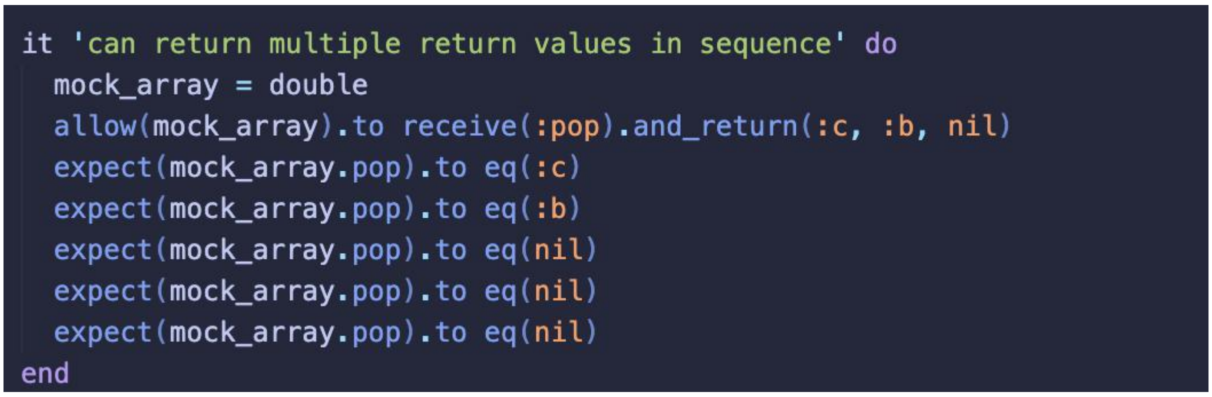
We only mock the sum method, all the other methods are valid still.



```ruby
it 'can stub one or more methods on a real object' do
  arr = [1, 2, 3]
  allow(arr).to receive(:sum).and_return(20)
  expect(arr.sum).to eq(20)
end
```

Setup multiple return values in sequence

Once will reach the last value, it will be repeated on the next calls.


```ruby
it 'can return multiple return values in sequence' do
  mock_array = double
  allow(mock_array).to receive(:pop).and_return(:c, :b, nil)
  expect(mock_array.pop).to eq(:c)
  expect(mock_array.pop).to eq(:b)
  expect(mock_array.pop).to eq(nil)
  expect(mock_array.pop).to eq(nil)
  expect(mock_array.pop).to eq(nil)
end
```

55. Matching Arguments

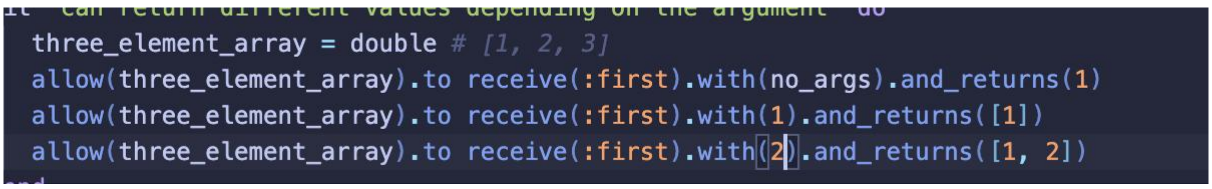
We have already seen

- A mockup method that returns an specific value
- A mockup method that returns a sequence of values

Now we are going to see how to return a different values depending on the number of parameters for example

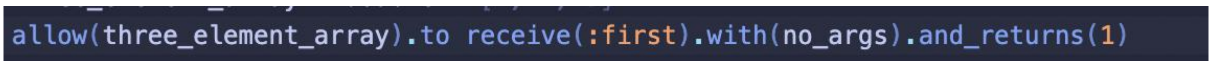
To do this we use method “**with**”

with:
You do not pass the number of arguments, you pass the value of the argument

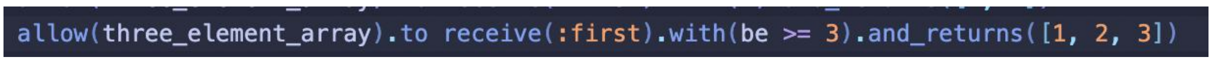

```ruby
it 'can return different values depending on the argument' do
  three_element_array = double # [1, 2, 3]
  allow(three_element_array).to receive(:first).with(no_args).and_returns(1)
  allow(three_element_array).to receive(:first).with(1).and_returns([1])
  allow(three_element_array).to receive(:first).with(2).and_returns([1, 2])
```

no_args

What to return when no args are being passed.


```ruby
allow(three_element_array).to receive(:first).with(no_args).and_returns(1)
```

Quantifier operator


```ruby
allow(three_element_array).to receive(:first).with(be >= 3).and_returns([1, 2, 3])
```

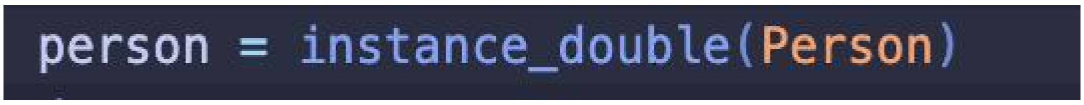
Resources

https://relishapp.com/rspec/rspec-mocks/v/3-8/docs/setting-constraints/matching-arguments

56. Instance Doubles

The doubles are great but put a lot of pressure into us (the developer) to try to mimic the real life class. Code changes frequently.

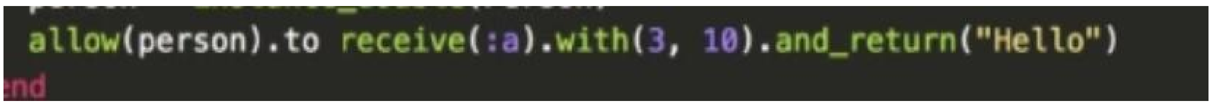
To create a double that only accepts the methods defined on an instance of a class we use "**instance_double**"



```
person = instance_double(Person)
```

Combined with allow and with

Remember that "allow and with" make you able to modify methods. If we used it with a instance_method, then if we try to configure the mockup method with the wrong number of arguments, then it is going to fail.

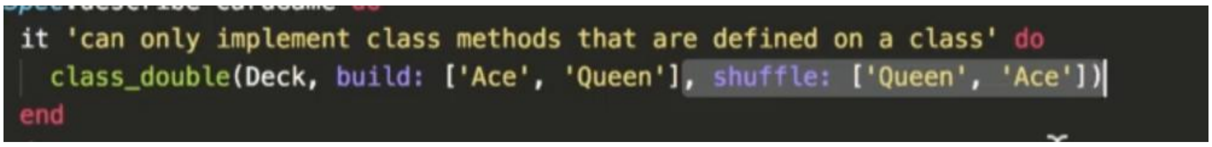


```
  allow(person).to receive(:a).with(3, 10).and_return("Hello")
nd
```

It is recommendable to use whenever it is possible to use instance_double

57. Class Doubles

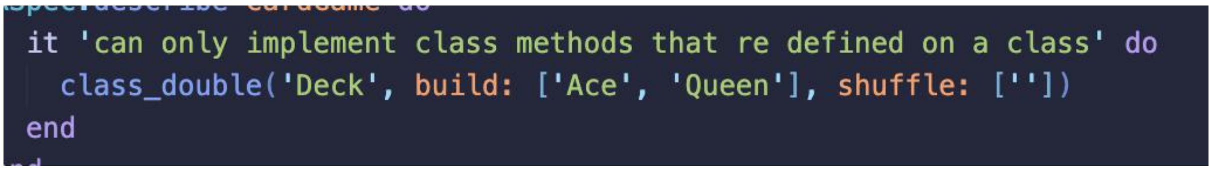
Similar to instance_double, but this one verifies the class methods of a class.



```
 it 'can only implement class methods that are defined on a class' do
   class_double(Deck, build: ['Ace', 'Queen'], shuffle: ['Queen', 'Ace'])
 end
```

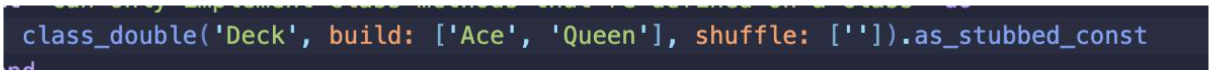
What if no class Deck exists yet?

We are creating a class_double of class "Deck" to test CardGame, the problem is that on a TDD approach, you probably just created the class "CardGame" but not the class "Deck". What if you want to create a Deck class double?. You should write the class name as a string.



```
  it 'can only implement class methods that re defined on a class' do
    class_double('Deck', build: ['Ace', 'Queen'], shuffle: [''])
  end
```

as_stubbed_const



```ruby
class_double('Deck', build: ['Ace', 'Queen'], shuffle: ['']).as_stubbed_const
```

Will make sure that on every Deck call from there to below, will replace it with the double. So if a class makes a call to Deck, it is going to be replaced automatically by the double.

Which one to use (instance or class double)

Depends if you want to mock instance methods or class methods.

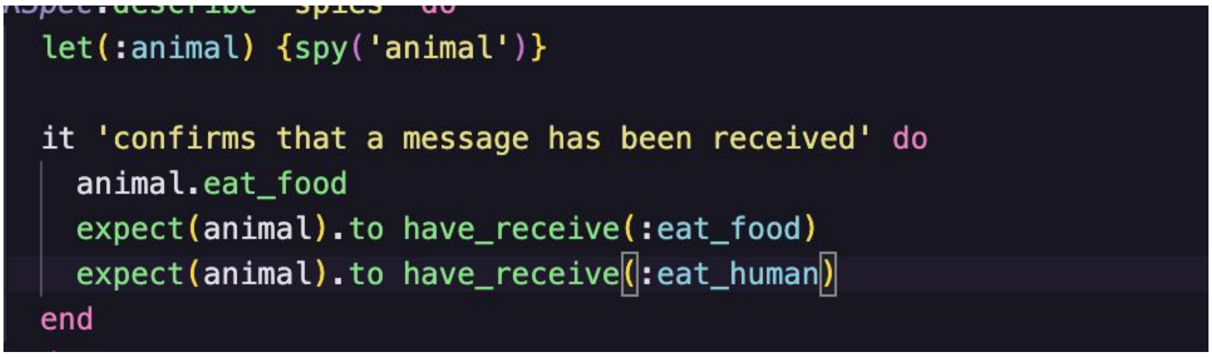
58. Spies I

There are differences between mock, double and spy, regularly developers use the terms interchangeable. You can do the same.

They are similar to doubles but:

- Doubles
  - Places expectation before the action
- Spy
  - Places the expectation after the action

The big advantage of spies is that they are going to absorb any message, even the ones that are not declared.



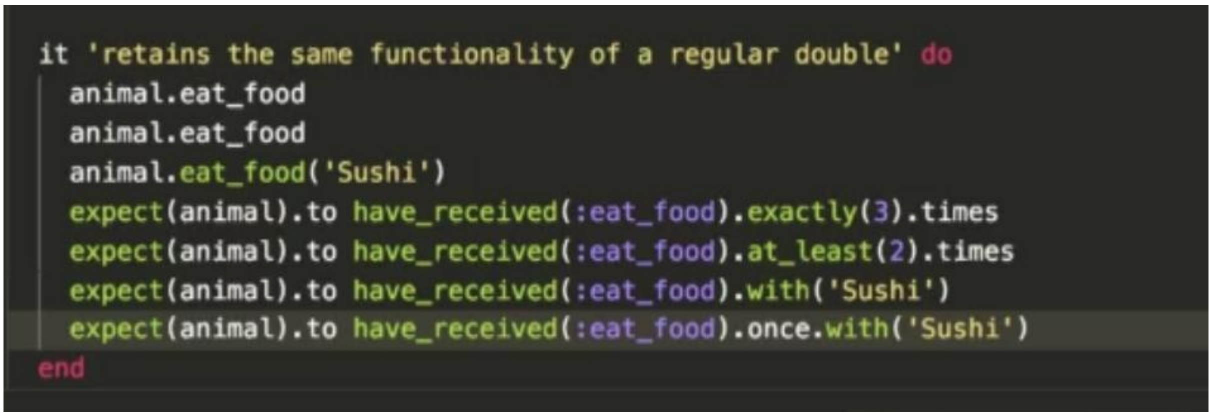
```ruby
  let(:animal) {spy('animal')}

  it 'confirms that a message has been received' do
    animal.eat_food
    expect(animal).to have_receive(:eat_food)
    expect(animal).to have_receive(:eat_human)
  end
```

As you can see:

- We did not declare the “eat food” and human methods but it is still valid.
- We first call the method and then we ask if the method was called.

Not the a big difference to a double

There are debates about which one is better, it is your preference about which on eto use.

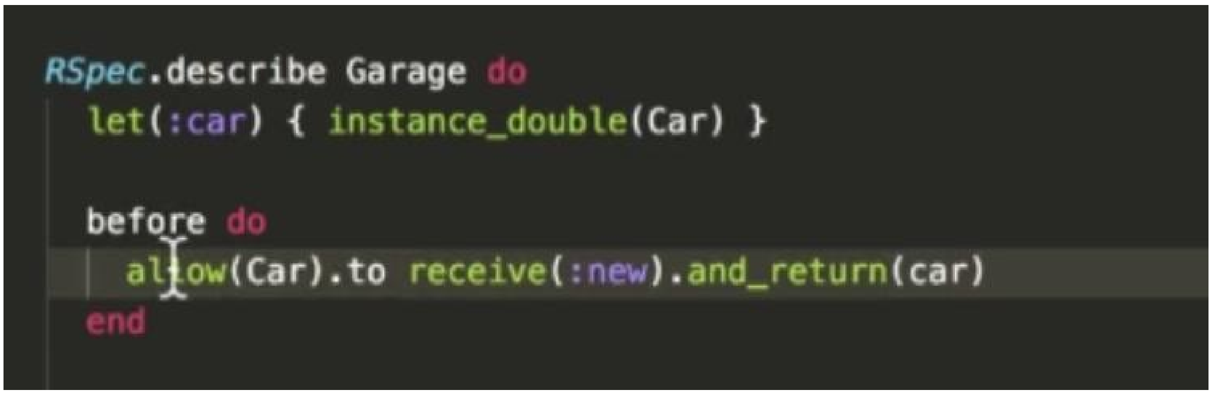


```ruby
it 'retains the same functionality of a regular double' do
  animal.eat_food
  animal.eat_food
  animal.eat_food('Sushi')
  expect(animal).to have_received(:eat_food).exactly(3).times
  expect(animal).to have_received(:eat_food).at_least(2).times
  expect(animal).to have_received(:eat_food).with('Sushi')
  expect(animal).to have_received(:eat_food).once.with('Sushi')
end
```

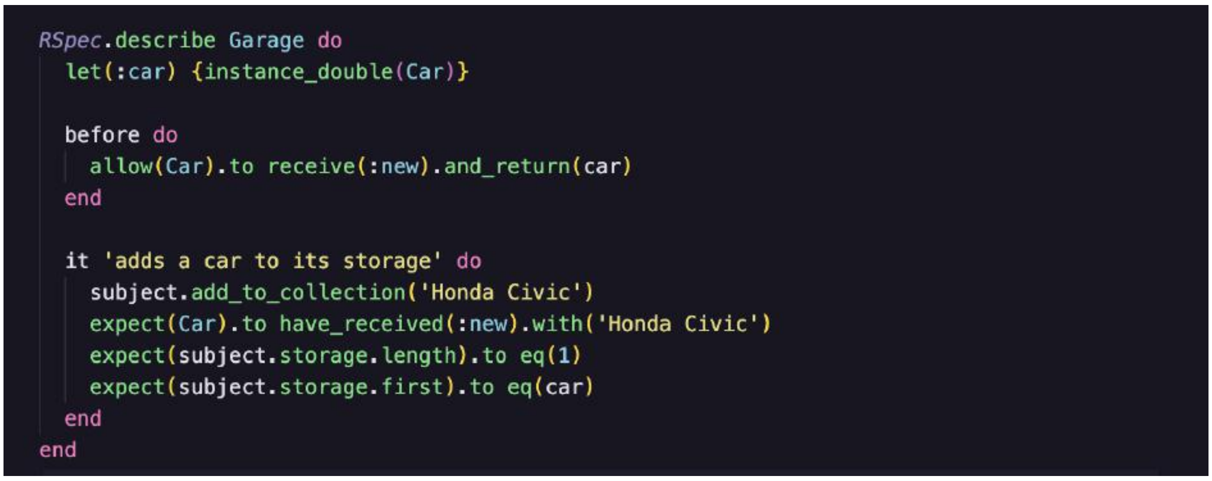
59. Spies II

We are going to fake the class and instance.
We are going to see that the **allow** method also enables us to have spy-like functionality whenever we mock up methods for a real ruby method. So we have to do the assertions after we call the method.

What we are doing is return an instance double whenever the new method is called.



```ruby
RSpec.describe Garage do
  let(:car) { instance_double(Car) }

  before do
    allow(Car).to receive(:new).and_return(car)
  end
```



```ruby
RSpec.describe Garage do
  let(:car) {instance_double(Car)}

  before do
    allow(Car).to receive(:new).and_return(car)
  end

  it 'adds a car to its storage' do
    subject.add_to_collection('Honda Civic')
    expect(Car).to have_received(:new).with('Honda Civic')
    expect(subject.storage.length).to eq(1)
    expect(subject.storage.first).to eq(car)
  end
end
```