Time	~ 2 - 3 hours
Learning Goals	• Understand the importance of writing tests • Learn how to write rspec tests

We're going to take a trip back in time - to the land of Ruby Todos. You are already familiar with the premise, and you have probably written solution code. But it is always tedious to test your code by clicking all features every time to make sure all codes are tested and run well.

In this challenge, you'll be writing the specs for a simplified version of the Ruby TODO challenge. Don't worry, you won't be testing the user interface. You'll just be writing unit tests.

A reminder on writing object-oriented specs

When you are writing your testing examples, keep in mind that writing a test is equivalent to adding dependencies on your code. As a general principle, you want to avoid dependencies.

To be both effective and well-designed, your specs need to enforce the contracts for your classes. They should still treat your objects with the same deference and respect that the other objects in your application code do.

You want your tests to be as loosely coupled to the application code as possible. By necessity, they need to know the name of the class, its methods, and the inputs and return values of those methods.

When writing unit tests, be wary of the temptation to test too deeply , i.e. to test the *implementation* instead of the *interface*. Remember, you don't care *how* your code works, you just want to ensure that it works as expected.

Objectives

Start the challenge by forking and cloning from this repository. You will find two application files: `list.rb` and `task.rb`, and two spec files `list_spec.rb` and `task_spec.rb`.

You can run the spec suite like this:

```
$ rspec task_spec.rb
# [test output]
$ rspec list_spec.rb
```

```
# [test output]
```

Read the code

Before you write any examples, be sure to read through all the codes and the specs.

The specs for `Task` were written but not yet for the `List`.

Write the specs

Once you understand how the code works, you can begin writing full specs for `List`. Your examples should test the full public interface of `List`.

Again, ensure that your specs actually test the code by checking that they fail when you comment out the relevant application code.

For reference, here are some good questions to ask when writing tests:

- What constitutes valid vs. invalid inputs?
- What are the expected and unexpected return values?
- What other effects does the method have on the state of the program?

Refactor to avoid errors

Over the course of writing your tests, you may find yourself discovering inconsistencies or irregularities in the code. That's great! That is what is supposed to happen! Writing specs are a great way to analyze your code more closely.

As you write the specs for `List` and come across an oversight in the program architecture, don't be afraid to make improvements to the code. You shouldn't change the public interface, but you can certainly change the implementation of the public methods.

As an example, let's look into the `#complete_task` method:

```
def complete_task(index)
  tasks[index].complete!
end
```

When you are writing the examples for this method, you should ask yourself: "What constitutes invalid input?" One of the answers is "when the `index` is beyond the scope of

the array". When this happens, `complete!` method will be implemented on `nil`, which is not good.

In this case, the spec has exposed a potential flaw in our program. It would be better to catch this potential error by refactoring the codes:

```
def complete_task(index)
 return false unless tasks[index]
 if tasks[index].complete!
  return true
 end
end
```

This code is an improvement in two ways:

1. Invalid input is caught and will return `false`, which we can now test for.
2. It no longer returns the result of method `#complete!`. Instead, it merely checks to be sure that when tasks[index] is valid(not nil), `#complete!` returns a value and returns boolean: true. This ensures that we have control over the return value of `#complete_task` regardless of what `#complete!` returns.

Different RSpec Syntax

The rspec-given gem was created to make test specifications more readable for humans.

`$gem install rspec-given` if this gem was not installed before. Under `task_spec.rb`, the final context can be written using rspec-given:

```
require "rspec-given"
require "rspec"
require_relative "task"


describe Task do
```

```
  Given(:description) { "Walk the giraffe" }
  Given(:task) { Task.new(description) }


  describe "#complete!" do
    When {task.complete!}
    Then {task.complete? == true}
  end
end
```