### 1. Install RSpec

RSpec doesn't come bundled with Ruby, so first we need to download it. The RubyGems package management system makes this trivial.

1. To install the version of RSpec we used in the video (2.8.0), which we recommend, hop over to a command line prompt and type
2. gem install rspec --version 2.8.0
3. This will pull down that specific version of the rspec gem and all of its dependencies from the [RubyGems remote repository](http://rubygems.org/).
4. If you'd prefer to use the very latest version of RSpec (3.0.0), then install it using
5. gem install rspec
6. To use RSpec 3 throughout the exercises, you'll need to [arrange things](https://online.pragmaticstudio.com/courses/ruby/steps/24#rspec3) a bit differently.
7. Then go ahead and list your locally installed gems by typing
8. gem list
9. You should see rspec and friends listed in the output, something like this:
10. \*\*\* LOCAL GEMS \*\*\*  
      
    rspec (2.8.0)  
    rspec-core (2.8.0)  
    rspec-expectations (2.8.0)  
    rspec-mocks (2.8.0)
11. Finally, just as a sanity check, use the rspec command to check the version and print out help information.
12. rspec --version  
    rspec --help
13. If both of those commands are successful, then you're good to go!

### 2. Write Player Examples

With RSpec installed, it's time to write code examples for the expectations we specified in the objective.

1. We want to test the behaviors (methods) that are contained in the Player class, which is defined in the player.rb file. By convention, the tests for that file need to be written in a file named player\_spec.rb. Go ahead and create a new file named player\_spec.rb in your studio\_game directory, and add the following code as a starting point:
2. require\_relative 'player'  
     
   describe Player **do**  
     
    *# examples go here*  
     
   **end**
3. Although this code may appear to be a specialized testing language of some sort, it's actually Ruby code. We call this a *domain-specific language* written in Ruby. The domain in this case is testing. Said another way, RSpec is a domain-specific language for specifying (testing) the behavior of objects. The vocabulary that RSpec uses is slightly different than what you may have heard in testing realms, so let's unpack what's going on here.
4. Every *spec file* has this same basic structure. In this case, the spec file will describe the expected behavior of a player. To do that, we'll need to make calls against the Player class. So the first thing we do is require the code in the player.rbfile.
5. Next, the describe method defines what RSpec calls an *example group*, which is just a convenient way to organize the spec file. The describe method either takes the name of the group as a string or, as we've done here, you can give it a class name. The describe method also takes a block (code between do and end). Inside the block you put the code examples that make up the group.
6. Now, back over at your command prompt, change into the studio\_game directory where the spec file lives. Then run that spec file with the rspec command, like so:
7. rspec player\_spec.rb
8. You should see output like this:
9. No examples found.  
     
     
   Finished in 0.08049 seconds  
   0 examples, 0 failures
10. ***TextMate Users:*** *If you want to run RSpec spec files in TextMate like we did in the video, you'll need to install the RSpec TextMate Bundle.*
11. ***Sublime Text Users:*** *We're not currently aware of an easy way to run RSpec files in Sublime Text, so we recommend simply running spec files from the command line instead.*
12. Obviously, the next step is to add a code example. A *code example* is an executable example of how the code is intended to be used and its expected behavior. Each code example follows a similar sequence of steps. It first invokes your code and gets back some results. It then checks that the results are what you expect.
13. Let's start with a simple one: a player should have a capitalized name. To express that as an RSpec code example, add the following code inside of the describe block:
14. it "has a capitalized name" **do**  
     player = Player.new("larry", 150)  
      
     player.name.should == "Larry"  
    **end**
15. Again, the language feels unfamiliar here, so let's break this down. The it method defines a code example, where the name of the example is an arbitrary string. Generally you want to use a string that expresses the behavior you're expecting. The it method takes a block as well, and inside *that* block you put the example code and *expectations*. An *expectation* is simply an expression of how you expect the code to behave.
16. For instance, this example code simply creates a new player object with a lowercase name. The expectation is that the name attribute should return the capitalized name. The expectation syntax is a bit funky. RSpec arranges things behind the scenes so that every object has a should method. It does that in an effort to make the expectations read more like English.
17. So all we're doing here is calling should on the result of calling the name attribute, which is a String object. Then we compare that result with the value we expect using the == equality operator. (If we mistakenly used =, it would be an assignment.) Here's the full spec file at this point:
18. require\_relative 'player'  
      
    describe Player **do**  
      
     it "has a capitalized name" **do**  
     player = Player.new("larry", 150)  
      
     player.name.should == "Larry"  
     **end**  
      
    **end**
19. **RSpec 3:** If you're using RSpec 3, then you may prefer to use the [new expectation syntax](https://online.pragmaticstudio.com/courses/ruby/steps/24#expect):
20. expect(player.name).to eq("Larry")
21. Now run the spec again:
22. rspec player\_spec.rb
23. It should pass with output like the following:
24. .  
      
    Finished in 0.08268 seconds  
    1 example, 0 failures
25. This is RSpec's default output format. Each dot at the top represents a passing example (only one in this case). In the summary line at the bottom you see that it ran one example, and it passed, so there are no failures. This is helpful. You can glance at that output and immediately know that things went well.
26. To add a splash of color to the output, run it with the --color option like this:
27. rspec player\_spec.rb --color
28. Now each passing example gets a *green* dot. And if all the examples pass the summary line is green, too. Green is good!
29. .  
      
    Finished in 0.08268 seconds  
    1 example, 0 failures
30. Great, now just to double-check that we're not getting a false positive, let's make the example fail. It's important to see a failing example; it means that the example is actually doing something useful.
31. Back over in your Player class, change the initialize method so that it doesn't capitalize the name. Yes, you're intentionally breaking the code to trigger a failing example. You'd never want to do this without tests!

def initialize (name, health=100)  
 @name = name  
 @health = health  
 end

1. Run the spec (in technicolor, of course) and it should fail with output like this:
2. F  
     
   Failures:  
     
    1) Player has a capitalized name  
    Failure/Error: player.name.should == "Larry"  
    expected: "Larry"  
    got: "larry" (using ==)  
     
   Finished in 0.07962 seconds  
   1 example, 1 failure
3. Cool! Seriously, this is good stuff. When the example ran, it caught the mismatched expectation. Each failing example (one in this case) is indicated by a red F. And if any of the examples fail, the summary line is red, too. More important, the output gives us a clue as to what went wrong. Notice that when an expectation fails, the expected and actual values are printed: it expected to get "Larry" but instead got "larry". That makes sense—the name is no longer being capitalized in our code.
4. Now fix the code! You know you're done when the spec passes again.

def initialize (name, health =100)  
 @name = name.capitalize  
 @health = health  
 end

1. That's a good start. Now we want to write examples for the other behaviors of the Player class. So let's express those as a list of *pending examples*. Add the following code inside the describe block, after your existing code example:
2. it "has an initial health"  
     
   it "has a string representation"  
     
   it "computes a score as the sum of its health and length of name"  
     
   it "increases health by 15 when w00ted"  
     
   it "decreases health by 10 when blammed"
3. Save your file, run the spec, and you should see output like this:
4. .\*\*\*\*\*  
     
   Pending:  
    Player has an initial health  
    # Not Yet Implemented  
    Player has a string representation  
    # Not Yet Implemented  
    Player computes a score as the sum of its health and length of name  
    # Not Yet Implemented  
    Player increases health by 15 when w00ted  
    # Not Yet Implemented  
    Player decreases health by 10 when blammed  
    # Not Yet Implemented  
     
   Finished in 0.07976 seconds  
   6 examples, 0 failures, 5 pending
5. Hey, that's pretty neat. By calling it without a block, RSpec prints the example as something we need to implement. Think of it as a to-do list for programmers. Notice that each pending example (5 in this case) is indicated by a yellow asterisk (\*). If there are no failures, but there are pending examples, the summary line is yellow.
6. Now that you have a list of pending examples, it's your turn to check them off the list. Simply turn all those yellow asterisks into green dots!
7. First, write an example that expects a player to have a health of 150 if it was created with an initial health value of 150.

it “has an initial health” do  
 player = Player.new(“larry”, 150)  
 player.health.should ==150  
 end

1. Don't forget to run the spec to get a reward of one more green dot!
2. Next, when you call to\_s on a player, we want to make sure it returns a properly formatted string. Write a code example for that behavior.

it “has a string representation” do  
 player = Player.new(“larry”, 150)  
 player.to\_s.should == “I’m Larry with a health of 150 and a score of 155”  
 end

1. The Player class has a score method that returns a score as the sum of the player's health and length of their name. Write a code example for that behavior.

it “computes a score as the sum of its health and length of name” do  
 player = Player.new(“larry”, 150)  
 player.score.should == (150+5)  
 end

1. When the player is w00ted, their health should increase by 15. Write another code example for that behavior.  
   it “increases health by 15 when w00ted” do  
    initial\_health = 150  
    player = Player.new(“larry”, initial\_health)  
    player.w00t  
     
    player.health.should == initlai\_health + 15  
   end
2. And when a player is blammed, their health should decrease by 10. Yup, write a spec for that, too.

it “decreases health by 10 when blammed” do  
 initial\_health = 150  
 player = Player.new(“larry”, initial\_health)  
 player.blam  
  
 player.health.should == initial\_health - 10  
 end

1. When you're all done, make sure to run the spec file one last time. You should see green across the board, like this:
2. ......  
     
   Finished in 0.0818 seconds  
   6 examples, 0 failures
3. Six green dots! Now isn't that satisfying?
4. *In addition to the dots, you'll likely see other output stating that the player got w00ted or blammed. We'll fix that in the Bonus Round.*

### 3. Remove Duplication

Notice that each code example has common setup code. Just like with non-test code, we don't want duplication in our test code. So after all the specs are passing, it's time to remove the duplication.

To share common setup code across code examples, RSpec provides a before method that runs once before *each* of the examples. Instance variables set in before are accessible in the code examples.

1. Create a before block that initializes an @initial\_health and @player instance variables.

before do  
 @initial\_health = 150  
 @player = Player.new(“larry”, @initial\_search)  
 end

1. Now, change all the code examples to use the instance variables and remove the duplicated lines of code.

it “has a capitalized name” do  
 @player.name.should == “Larry”  
 end  
   
 it “has an initial health” do  
 @player.health.should == 150  
 end  
   
 it “has a string representation” do  
 @player.to\_s.should == “I’m Larry with a health of 150 and a score of 155.”  
 end  
  
 it “computes a score as the sum of its health and length of name” do   
 @player.score.should == (150 + 5)  
 end  
  
 it “increase health by 15 when w00ted” do  
 @player.w00t  
 @player.health.should == @initial\_health + 15  
 end  
  
 it “decreases health by 10 when blammed” do  
 @player.blam  
 @player.health.should == @initial\_health - 10  
 end

1. Make sure to run the specs to make sure they all still pass!

### Solution

The full solution for this exercise is in the testing directory of the [code bundle](https://s3.amazonaws.com/pragmaticstudio/courses/ruby/pragstudio-ruby-code.zip).

### Bonus Round

#### Tweak the Output

You may have noticed that when you run the specs, you see output interspersed with the dots, like this:

....Larry got w00ted!  
.Larry got blammed!  
.  
  
Finished in 0.0818 seconds  
6 examples, 0 failures

Where's that extra output coming from? Well, remember that your w00t and blam methods use puts to print what happened to the player to the console. Normally a Game object calls these methods when you run the game program. But in the spec we're calling w00t and blam directly, so when the spec runs we see the results of the puts interspersed with the spec results. In other words, RSpec doesn't suppress standard output.

So when the spec runs, we need a way to "turn off" standard output. Thankfully, Ruby makes that easy by providing access to the standard output stream via the global $stdout variable (global variables always start with $). By default, $stdout is set to an input/output object that prints to the console. But we can override that by assigning a different object to $stdout. For example, a StringIO object acts just like other I/O objects, but it reads and writes to strings rather than the console. A simple workaround to our problem then is to set the global $stdout variable to a new StringIO object like so:

$stdout = StringIO.new

That way, standard output will be written to the StringIO object instead of to the console. Now, we don't want to override standard output when the game program runs. Rather, we only want to do this when the specs run—specifically before each code example runs. Change your player\_spec.rb file to use this workaround.

before do  
 $stdout = StringIO.new  
end

Now when you run the spec, you should only see green dots.

#### Fundraising Program

Are you still working on the fundraising program? We hope so. Ready to write some tests for it? Yes! Below are a few ideas to get you started.

Write expectations for the following behaviors of a project:

* has an initial target funding amount
* computes the total funding outstanding as the target funding amount minus the funding amount
* increases funds by 25 when funds are added
* decreases funds by 15 when funds are removed
* has a default value of 0 for funding amount

Make sure you see all green dots before moving on.

### crowdfund.rb

require\_relative 'project'

require\_relative 'fundrequest'

project1 = Project.new("Project ABC", 5000, 1000)

project2 = Project.new("Project LMN", 3000, 500)

project3 = Project.new("Project XYZ", 75, 25)

projects = FundRequest.new("VC-Friendly Start-up Projects")

puts projects.title

projects.add\_project(project1)

projects.add\_project(project2)

projects.add\_project(project3)

projects.request\_funding

fundrequest.rb

require\_relative 'project'

class FundRequest

attr\_reader :title

def initialize(title)

@title = title

@projects = []

end

def add\_project(a\_project)

@projects.push(a\_project)

end

def request\_funding

puts "There are #{@projects.size} projects that you could fund:"

@projects.each do |project|

puts project

end

@projects.each do |project|

project.add\_funds

project.remove\_funds

project.add\_funds

puts project

end

end

end

project.rb

class Project

attr\_accessor :name

attr\_reader :funding, :target

def initialize(name, target\_funding\_amount, funding=0)

@name = name

@target = target\_funding\_amount

@funding = funding

end

def to\_s

"#{@name} has $#{@funding} in funding towards a goal of $#{@target}."

end

def remove\_funds

@funding -= 15

puts "#{@name} lost some funds!"

end

def add\_funds

@funding += 25

puts "#{@name} got more funds!"

end

def total\_funding\_outstanding

@target - @funding

end

end

if \_\_FILE\_\_ == $0

project = Project.new("Project ABC", 5000, 1000)

puts project.name

puts project.funding

project.remove\_funds

puts project.funding

project.add\_funds

puts project.funding

end

project\_specifications.rb

require\_relative 'project'

describe Project do

before do

@initial\_funds = 1000

@project = Project.new("Project ABC", 5000, @initial\_funds)

$stdout = StringIO.new

end

it "has an initial target funding amount" do

@project.target.should == 5000

end

it "computes the total funds outstanding as the target funding amount minus the funding amount" do

@project.total\_funding\_outstanding.should == (5000 - 1000)

end

it "increases funds by 25 when funds are added" do

@project.add\_funds

@project.funding.should == @initial\_funds + 25

end

it "decreases funds by 15 when funds are removed" do

@project.remove\_funds

@project.funding.should == @initial\_funds - 15

end

context "created without a funding amount" do

before do

@project = Project.new("Project ABC", 5000)

end

it "has a default funding amount of 0" do

@project.funding.should == 0

end

end

end

### RSpec 3

With the recent release of RSpec 3, you'll get a deprecation warning if you use the traditional should syntax that we use throughout this course. The newer expect syntax is now the default (see examples below), though we still prefer the traditional syntax. Thankfully, there are no plans to remove the should syntax from RSpec. However, you'll need to explicitly enable it in order to silence the deprecation warnings.

So if you use RSpec 3, you will either need to explicitly enable the should syntax or update your specs to use the newer expect syntax.

#### Enable should Syntax

To enable the traditional should syntax, create a file named spec\_helper.rb in your studio\_game directory and paste the following code into it:

RSpec.configure **do** |config|  
 config.expect\_with **:rspec** **do** |c|  
 c.syntax = [**:should**, **:expect**]  
 **end**  
 config.mock\_with **:rspec** **do** |c|  
 c.syntax = [**:should**, **:expect**]  
 **end**  
**end**

This enables both the use of the traditional should syntax, as well as the newer expect syntax.

Then, at the top of your player\_spec.rb file, add the following line which loads your spec\_helper.rb file:

require\_relative 'spec\_helper'

You should then be able to run your specs and not see any deprecation warnings. :-)

#### Use the New Expectation Syntax

The traditional expectation syntax calls the should method on the object we're testing and then compares that object with the value we expect using == equality operator, like so:

@player.name.should == "Larry"

By comparison, the same expectation expressed in the newer style uses the expect method and looks like this:

expect(@player.name).to eq("Larry")

This new syntax looks different, but it's essentially doing the same thing. The expect method takes the object we're interested in testing. In this case, we're writing an expectation for reading the name attribute of the @player object, so that's what we pass as the parameter to the expect method. After calling expect we tack on a call to the to method (or not\_to method). The to method takes what RSpec calls a *matcher* that performs some type of comparison and returns true or false. If the matcher returns true the expectation passes and if it returns false the expectation fails. In this example we use the eqmatcher which will return true if the player's name equals "Larry" and false otherwise. If you step back and squint slightly, that line kinda reads like English: "Expect the player's name to equal 'Larry'."

If you prefer this new expectation syntax, feel free to change your player\_spec.rb file to use it. Throughout the exercises we'll continue to use the traditional should syntax for consistency.

it "has a capitalized name" do  
 expect(@player.name).to eq("Larry")  
end

it "has an initial health" do  
 expect(@player.health).to eq(150)  
end

it "has a string representation" do  
 expect(@player.to\_s).to eq("I'm Larry with a health of 150 and a score of 155.")  
end

it "computes a score as the sum of its health and length of name" do  
 expect(@player.score).to eq(150 + 5)  
end

it "increases health by 15 when w00ted" do  
@player.w00t

expect(@player.health).to eq(@initial\_health + 15)  
end

it "decreases health by 10 when blammed" do  
 @player.blam  
 expect(@player.health).to eq(@initial\_health - 10)  
end

### Wrap Up

Testing is an integral part of writing good Ruby code. At first, it might feel like writing tests slows you down. But in the long run, tests are a huge time-saver. They help you find (and fix) bugs early, and tests give you the confidence to refactor (change) code without the fear of breaking existing functionality.

Get into the habit of writing a little code, then writing a test or two to verify your expectations, and then writing a little more code. It takes practice, but it will serve you well.

In this exercise, you learned how to:

* install a gem (RSpec)
* create a spec file
* use example groups to organize a spec file
* write executable code examples with expectations for how your code should behave
* manage a to-do list with pending examples
* run automated specs

With some tests in place, are you ready to take the game up a notch? Right now the outcome of playing a game is totally predictable: Curly always wins! In the next section, we'll use conditionals and loops in the game to *randomly* w00t or blam the players. We'll also create a new class and (you guessed it) specify its expected behavior using RSpec.