Basic RSpec Structure

> describe

`describe` accepts a string or class. It is used to organize specs.

```
describe User do
end

describe "a user who has admin access" do
end
```

> it

`it` is what describes the spec. It optionally takes a string.

```
describe User do
  it "generates an authentication token when created" do
  end

  it { }
end
```

> should

`should` is what RSpec uses to make assertions.

```
describe Array do
  it "reports a length of zero without any values" do
    [].length.should == 0
  end
end
```

> should_not

`should_not` is the inverse of `should`.

```
describe Array, "with items" do
  it "reports a length of anything other than zero" do
    [1, 2, 3].length.should_not == 0
  end
end
```

Variables Available to Specs

> subject

`subject` helps signify what's being tested.

```
describe User, "with admin access" do
  subject { User.create(:admin => true, :name => "John Doe") }

  it "displays its admin capabilities in its name" do
    subject.display_name.should == "John Doe (admin)"
  end
end
```

When `describe` is passed a class, `subject` is implied as `described_class.new`.

```
describe Array do
  # subject { described_class.new } # implied because subject has not been defined

  it "has a length of zero" do
    subject.length.should == 0
  end
end
```

Callbacks

> before

`before` runs the specified block before each test. Useful for preparing data for each test in the `describe` block.

```
describe User, "with friends" do
  subject { User.new }

  before do
    subject.friends += [ Friend.new, Friend.new ]
  end

  it "counts friends correctly" do
    subject.friends.length.should == 2
  end
end
```

> after

`after` runs the specified block after each test. Useful for cleaning up data or side-effects of a test run.

```
describe ReportGenerator, "generating a PDF" do
  after do
    ReportGenerator.cleanup_generated_files
  end

  it "includes the correct data" do
    data = [ [1,2], [3,2], [4,1] ]
    ReportGenerator.generate_pdf(data).points.length.should == 3
  end
end
```

> around

`around` runs the specified code around each test. To execute the test, call `run` on the block variable.

```
describe ReportGenerator, "with a custom PDF builder" do
  around do |example|
    default_pdf_builder = ReportGenerator.pdf_builder
    ReportGenerator.pdf_builder = PdfBuilderWithBorder.new("#000000")
    example.run
    ReportGenerator.pdf_builder = default_pdf_builder
  end

  it "adds a border to the PDF" do
    data = [ [1,2], [3,2], [4,1] ]
    ReportGenerator.generate_pdf(data).border_color.should == "#000000"
  end
end
```

This is a great way to test overriding class attributes (which can, among other things, be used for dependency injection) by storing the old value, overriding it, and then reassigning it after the test is complete.

Test Optimizations

> let

`let` lazily-evaluates a block and names it after the symbol. This is another way to help display the intent of your specs.

```
describe User, "with friends" do
  let(:friends) { [Friend.new, Friend.new] }
  subject       { User.with_friends(friends) }

  it "keeps track of friends correctly" do
    subject.friends.should == friends
  end
end
```

> def your_own_method

Define your own methods to use within the context of the `describe` block. Another way to simplify tests by displaying intent with method names.

```
describe InvitationMailer do
  def deliver_email
    from_user = User.new(:email => "sender@example.com")
    to_user   = User.new(:email => "recipient@example.com")
    InvitationMailer.invitation(from_user, to_user).deliver
    yield from_user, to_user
  end

  it "delivers email from the sender to the receiver" do
    deliver_email do |from_user, to_user|
      to_user.should have(1).email.from(from_user)
    end
  end
end
```

Things to Avoid in RSpec

> its

`its` accepts a method (as a symbol) and a block, executing the method and performing an assertion on the result.

```
describe User, "with admin access" do
  subject            { User.create(:admin => true, :name => "John Doe") }
  its(:display_name) { should == "John Doe (admin)" }
end
```

While this looks pretty nice, pay attention to the behavior: for each `its`, the subject is mutating!

> let!

`let!` behaves like `let` but is not lazily-evaluated.

```
describe User, "with admin access" do
  let!(:friends) { [Friend.new, Friend.new] }
  subject        { User.with_friends(friends) }

  it "keeps track of friends correctly" do
    subject.friends.should == friends
  end
end
```

This will explicitly set up data for each test; expensive operations will slow down the test suite and this is never really necessary.