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
```

> expect().to

`expect().to` is RSpec's assertion syntax.

```
describe Array do
  it 'reports a length of zero without any values' do
    expect([].length).to eq 0
  end
end
```

> expect().not_to

`expect().not_to` is the inverse of `expect().to`.

```
describe Array, 'with items' do
  it 'reports a length of anything other than zero' do
    expect([1, 2, 3].length).not_to eq 0
  end
end
```

Callbacks

> before

`before` runs the specified block before each test. Often encourages bad tests.

```
describe User, 'with friends' do
  subject { User.new }
  before { subject.friends += [ Friend.new, Friend.new ] }

  it 'counts friends' do
    expect(subject.friends.length).to eq 2
  end
end
```

> after

`after` runs the specified block after each test. Typically unnecessary.

```
describe ReportGenerator, 'generating a PDF' do
  after { ReportGenerator.cleanup_generated_files }

  it 'includes the correct data' do
    expect(ReportGenerator.generate_pdf([1, 2, 3]).points.length).to eq 3
  end
end
```

> around

`around` runs the specified code around each test. To execute the test, call `run` on the block variable. Useful for `class_attribute` dependency injection.

```
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
    expect(ReportGenerator.generate_pdf([]).border_color).to eq '#000000'
  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 eq 'John Doe (admin)' }
end
```

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

> let

`let` lazily-evaluates a block and names it after the symbol. It often leads to "mystery guest" and "general fixture".

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

  it 'keeps track of friends correctly' do
    expect(subject.friends).to eq friends
  end
end
```

> let!

`let!` behaves like `let` but is not lazily-evaluated (it runs regardless if the spec uses it).

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

  it 'keeps track of friends correctly' do
    expect(subject.friends).to eq 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.

> subject

`subject` helps signify what's being tested but can lead to "mystery guest" or encouraging other bad habits like `before` blocks.

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

  it "displays its admin capabilities in its name" do
    expect(subject.display_name).to eq 'John Doe (admin)'
  end
end

describe User, 'without admin access' do
  subject { User.create(admin: false, name: 'John Doe') }

  it 'does not display its admin capabilities in its name' do
    expect(subject.display_name).to eq 'John Doe'
  end
end
```

Alternative Solutions for Things to Avoid

Inline Code in the Test

Here's an alternate implementation to using `subject` and `let` (or `before`); we build the list of friends and the user within the test, making it immediately obvious which variables are used.

```
describe User do
  it 'keeps track of friends correctly' do
    friends = [Friend.new, Friend.new]
    user = User.with_friends(friends)
    expect(user.friends).to eq friends
  end
end
```

Extract Helper Methods

Here's an alternate implementation to using `subject`; we build the object instance within the test, extracting a method which generates a user with the attributes assigned.

```
describe User, '#display_name' do
  it 'displays its admin capabilities in its name when an admin' do
    user = build_user name: 'John Doe', admin: true
    expect(user.display_name).to eq 'John Doe (admin)'
  end

  it 'displays no additional data when not an admin' do
    user = build_user name: 'John Doe', admin: false
    expect(user.display_name).to eq 'John Doe'
  end

  def build_user(options)
    User.new(options)
  end
end
```

Test Optimizations

Extract Complex Helper Methods

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

  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
end
```