**Table of Contents** (left)

# rspec-core

```
gem install rspec      # for rspec-core, rspec-expectations, rspec-mocks
gem install rspec-core # for rspec-core only
```

# overview

rspec-core provides the structure for writing executable examples of how your code should behave. It uses the words "describe" and "it" so we can express concepts like a conversation:

```
"Describe an order."
"It sums the prices of its line items."
```

## basic structure

```
describe Order do
  it "sums the prices of its line items" do
    order = Order.new
    order.add_entry(LineItem.new(:item => Item.new(
      :price => Money.new(1.11, :USD)
    )))
    order.add_entry(LineItem.new(:item => Item.new(
      :price => Money.new(2.22, :USD),
      :quantity => 2
    )))
    order.total.should eq(Money.new(5.55, :USD))
  end
end
```

The `describe` method creates an ExampleGroup. Within the block passed to `describe` you can declare examples using the `it` method.

Under the hood, an example group is a class in which the block passed to `describe` is evaluated. The blocks passed to `it` are evaluated in the context of an *instance* of that class.

## nested groups

You can also declare nested nested groups using the `describe` or `context` methods:

```
describe Order do
  context "with no items" do
    it "behaves one way" do
      # ...
    end
  end

  context "with one item" do
    it "behaves another way" do
      # ...
    end
  end
end
```

## aliases

You can declare example groups using either `describe` or `context`, though only `describe` is available at the top level.

You can declare examples within a group using any of `it`, `specify`, or `example`.

## shared examples and contexts

Declare a shared example group using `shared_examples`, and then include it in any group using `include_examples`.

```
shared_examples "collections" do |collection_class|
  it "is empty when first created" do
    collection_class.new.should be_empty
  end
end

describe Array do
  include_examples "collections", Array
end

describe Hash do
  include_examples "collections", Hash
end
```

Nearly anything that can be declared within an example group can be declared within a shared example group. This includes `before`, `after`, and `around` hooks, `let` declarations, and nested groups/contexts.

You can also use the names `shared_context` and `include_context`. These are pretty much the same as `shared_examples` and `include_examples`, providing more accurate naming for in which you share hooks, `let` declarations, helper methods, etc, but no examples.

## metadata

rspec-core stores a metadata hash with every example and group, which contains like their descriptions, the locations at which they were declared, etc, etc. This hash powers many of rspec-core's features, including output formatters (which access descriptions and locations), and filtering before and after hooks.

Although you probably won't ever need this unless you are writing an extension, you can access it from an example like this:

```
it "does something" do
  example.metadata[:description].should eq("does something")
end
```

### `described_class`

When a class is passed to `describe`, you can access it from an example using the `described_class` method, which is a wrapper for `example.metadata[:described_class]`.

```
describe Widget do
  example do
    described_class.should equal(Widget)
  end
end
```

This is useful in extensions or shared example groups in which the specific class is unknown. Taking the shared examples example from above, we can clean it up a bit using `described_class`:

```
shared_examples "collections" do
  it "is empty when first created" do
    described.new.should be_empty
  end
end

describe Array do
  include_examples "collections"
end

describe Hash do
  include_examples "collections"
end
```

## the `rspec` command

When you install the rspec-core gem, it installs the `rspec` executable, which you'll use to run rspec. The `rspec` comes with many useful options. Run `rspec --help` to see the complete list.

## see also

- http://github.com/rspec/rspec
- http://github.com/rspec/rspec-expectations
- http://github.com/rspec/rspec-mocks

## get started

Start with a simple example of behavior you expect from your system. Do this before you write any implementation code:

```
# in spec/calculator_spec.rb
describe Calculator do
  it "add(x,y) returns the sum of its arguments" do
    Calculator.new.add(1, 2).should eq(3)
  end
end
```

Run this with the rspec command, and watch it fail:

```
$ rspec spec/calculator_spec.rb
./spec/calculator_spec.rb:1: uninitialized constant Calculator
```

Implement the simplest solution:

```
# in lib/calculator.rb
class Calculator
  def add(a,b)
    a + b
  end
end
```

Be sure to require the implementation file in the spec:

```
# in spec/calculator_spec.rb
# - RSpec adds ./lib to the $LOAD_PATH
require "calculator"
```

Now run the spec again, and watch it pass:

```
$ rspec spec/calculator_spec.rb
.

Finished in 0.000315 seconds
1 example, 0 failures
```

Use the `documentation` formatter to see the resulting spec:

```
$ rspec spec/calculator_spec.rb --format doc
Calculator add
  returns the sum of its arguments

Finished in 0.000379 seconds
1 example, 0 failures
```

## Also see

- http://github.com/rspec/rspec
- http://github.com/rspec/rspec-expectations
- http://github.com/rspec/rspec-mocks