RSpec Cheatsheet

The `expect` syntax in rSpec 2.11 obsoletes `should` and `should_not`, and should be used for any new code. Behaviour is asserted by pairing `expect().to` and `expect().not_to` with a Matcher predicate.

Object predicates

Examples:
```
expect(a_result).to     eq("this value")
expect(a_result).not_to eq("that value")
```

Equality and Identity
```
eq(expected)    # same value
eql(expected)   # same value and type
equal(expected) # same object
```

True/False/nil
```
be_true   # true-ish
be_false  # false-ish
be_nil    # is nil
```

Numeric comparisons
```
be >= 10  # also applicable for >, <=, <
be_within(0.01).of(28.35) # floating point
```

Regex pattern matching
```
match /a regex/
```

Array and string prefixes/suffixes
```
start_with "free"
start_with [1,2,3]
end_with "dom"
end_with [3,4,5]
```

Array matching
Compares arrays for exact equivalence, ignoring ordering.
```
match_array [a,b,c]
match_array [b,c,a] # same result
```

Ancestor Class
```
be_a <class>  # or...
be_an <class>
be_a_kind_of <class>  # or...
be_kind_of <class>
be_an_instance_of <class>  # or...
be_instance_of <class>
```

Collection Size
When the target is a collection, "things" may be anything.
If the target owns a collection, "things" must be the name of the collection.
```
have(<n>).things
have_at_least(<n>).things
have_at_most(<n>).things
```

Containment and coverage
```
expect("string").to include "str"
expect([1,2,3]).to include 2,1
expect(1..5).to cover 3,4,5
```

Duck Typing
```
respond_to(:foo)
respond_to(:foo, :and_bar, :and_baz)
respond_to(:foo).with(1).argument
respond_to(:foo).with(2).arguments
```

Block predicates

Examples:
```
expect { raise "oops" }.to   raise_error
expect { some block }.not_to throw_symbol
```

Raising
`error` and `exception` are functionally interchangeable, so you're free to use whichever option best suits your context.
```
raise_error
raise_error RuntimeError
raise_error "the exact error message"
raise_error /message$/  # regexp
raise_error NameError, "exact message"
raise_error NameError, /error message/
```

Throwing
```
throw_symbol
throw_symbol :specificsymbol
throw_symbol :specificsymbol, with_arg
```

Yielding
```
yield_control
yield_with_args "match foo", /match bar/
yield_with_no_args
yield_successive_args "foo", "bar"
```

Changing
```
change{Counter.count}
change{Counter.count}.from(0).to(1)
change{Counter.count}.by(2)
```

Satisfying
`satisfy` is valid for objects and blocks, and allows the target to be tested against an arbitrarily specified block of code.
```
expect(20).to satisfy { |v| v % 5 == 0 }
```

Migrating from `should` to the new `expect` syntax

The shift from `should` to `expect` makes much of RSpec's code much cleaner, and unifies some aspects of testing syntax.

For the vast majority of the RSpec tests you're likely to write, the following examples will suffice to get you converted from `should` to `expect`.

```
Old: my_object.should eq(3)
New: expect(my_object).to eq(3)

Old: my_object.should_not_be_a_kind_of(Foo)
New: expect(my_object).not_to be_a_kind_of(Foo)
```

It should be noted that the syntax for mock objects has not yet been finalised. It will also begin to use `expect` in the near future, but for now `should` is still in use.

Anchor, 2013 — Based on the original by Dave Astel, 2006

Stubs and Mock objects

Creation of mock objects now uses the `double` method instead of `mock`. There are also plans to move to `expect` for defining the behaviour of mock objects, but this hasn't yet been finalised. Stubs remain unchanged.

Mocking a database connection that's expected to run a few queries:

```
test_db = double("database")
test_db.should_receive(:connect).once
test_db.should_receive(:query).at_least(3).times.and_return(0)
test_db.should_receive(:close).once
```

Using a stub in place of a live call to fetch information, which may be very slow:

```
world = World.new()
world.stub(:get_current_state).and_return( [1,2,3,4,5] )
```

Mocked behaviour

Creating a Double

```
foo = double(<name>)
foo = double(<name>, <options>)
# Currently a single option is supported:
foo = double("Foo", :null_object => true)
```

Expecting messages

```
double.should_receive(:<message>)
double.should_not_receive(:<message>)
```

Expecting arguments to messages

```
should_receive(:foo).with(<args>)
should_receive(:foo).with(:no_args)
should_receive(:foo).with(:any_args)
```

Defining explicit response of a double

```
double.should_receive(:msg) { block_here }
```

Arbitrary argument handling

```
double.should_receive(:msg) do | arg1 |
   val = do_something_with_argument(arg1)
   expect(val).to eq(42)
end
```

Receive counts

```
double.should_receive(:foo).once
         .twice
         .exactly(n).times
         .any_number_of_times
         .at_least(:once)
         .at_least(:twice)
         .at_least(n).times
```

Return values

```
should_receive(:foo).once.and_return(v)
      .and_return(v1, v2, ..., vn)
         # implies consequtive returns
      .at_least(n).times
      .and_return {...}
```

Raising, Throwing and Yielding

```
.and_raise(<exception>)
.and_throw(:symbol)
.and_yield(values, to, yield)
#  and_yield can be used multiple times for methods
#  that yield to a block multiple times
```

Enforcing Ordering

```
.should_receive(:flip).once.ordered
.should_receive(:flop).once.ordered
```

Stubs

Methods can be stubbed out on both doubles (mock objects) and real objects. Stubs are functionally similar to the use of `should_receive` on a double, the difference is in your intents.

Creating a Stub

All three forms are equally valid on doubles and real objects.

```
double.stub(:status) { "OK" }
object.stub(:status => "OK" )
object.stub(:status).and_return("OK")
```

Double with stub at creation time

```
double("Foo", :status => "OK")
double(:status => "OK") # name is optional
```

Multiple consecutive return values

A stubbed method can return different values on subsequent invocations. For any calls beyond the number of values provided, the last value will be used.

```
die.stub(:roll).and_return(1,2,3)
```

Example:

```
die.roll # returns 1
die.roll # returns 2
die.roll # returns 3
die.roll # returns 3
die.roll # returns 3
```

Raising, Throwing and Yielding

Stubs support `and_raise`, `and_throw` and `and_yield` the same was as doubles do. The syntax for use on stubs is identical.

Configuring RSpec with `spec_helper.rb`

The convention for configuring RSpec is a file named `spec_helper.rb` in your `spec` directory. It's always in your load path, so you `require 'spec_helper'` in each file.

This is the perfect place to enable coloured output, randomise the order that specs are run in, and apply formatters as appropriate.

```
RSpec.configure do |config|
   config.color_enabled = true
   config.order = "random"

   # This is critical, don't remove it
   config.formatter = 'NyanCatWideFormatter'
end
```

Perfect!

Anchor, 2013 — Based on the original by Dave Astel, 2006