**ATDD**

Web Test

Automation

**Table of Contents**

[TypeScript Cheat Sheet 3](#_Toc82386653)

[Comments 3](#_Toc82386654)

[Basic Types 3](#_Toc82386655)

[Declarations 4](#_Toc82386656)

[Variable Assignment 4](#_Toc82386657)

[Function Calls 4](#_Toc82386658)

[Type Assertions 5](#_Toc82386659)

[Interfaces 5](#_Toc82386660)

[Type Aliases 6](#_Toc82386661)

[Function Types 6](#_Toc82386662)

[Classes 6](#_Toc82386663)

[Inheritance 6](#_Toc82386664)

[Generics 7](#_Toc82386665)

[Conditionals (if) 7](#_Toc82386666)

[Arrays 8](#_Toc82386667)

[Iteration 8](#_Toc82386668)

[Truth and Falsehood 9](#_Toc82386669)

[Blocks 9](#_Toc82386670)

[Dictionaries 10](#_Toc82386671)

[Expression Substitution 10](#_Toc82386672)

[WATiR Cheat Sheet 11](#_Toc82386673)

[Browser 11](#_Toc82386674)

[Textbox 11](#_Toc82386675)

[Text Area 11](#_Toc82386676)

[Button 12](#_Toc82386677)

[Checkbox 12](#_Toc82386678)

[Listbox 12](#_Toc82386679)

[Radio 13](#_Toc82386680)

[Image 13](#_Toc82386681)

[Div 13](#_Toc82386682)

[Table 14](#_Toc82386683)

[General 14](#_Toc82386684)

[Waiting 15](#_Toc82386685)

[PageObject Cheat Sheet 16](#_Toc82386686)

[Create your page 16](#_Toc82386687)

[Describe your page 16](#_Toc82386688)

[Use your page 16](#_Toc82386689)

[Cucumber Cheat Sheet 18](#_Toc82386690)

[Scenario 18](#_Toc82386691)

[Steps 18](#_Toc82386692)

[Given 18](#_Toc82386693)

[When 19](#_Toc82386694)

[Then 19](#_Toc82386695)

[Step Arguments 19](#_Toc82386696)

[Data Tables 20](#_Toc82386697)

[Scenario Outline 20](#_Toc82386698)

[Tags 21](#_Toc82386699)

[Reports 21](#_Toc82386700)

[Text 22](#_Toc82386701)

[Calling a step or steps from within a step 22](#_Toc82386702)

[Calling steps 22](#_Toc82386703)

[Calling steps that have a Data Table as an argument 22](#_Toc82386704)

[Cucumber Configuration 23](#_Toc82386705)

[Defining Profiles 23](#_Toc82386706)

[Executing Profiles 23](#_Toc82386707)

[Default Profile 23](#_Toc82386708)

[RSpec Cheat Sheet 25](#_Toc82386709)

[Object predicates 25](#_Toc82386710)

[Block predicates 26](#_Toc82386711)

[Appendix 28](#_Toc82386712)

# TypeScript Cheat Sheet

## Comments

// Comment  
  
/\*   
 Multiline Comment  
\*/

## Basic Types

Any  
void  
  
Boolean  
number  
string  
  
null  
  
undefined  
  
string[] // or Array<string>  
[string, number] // tuple  
string | null | undefined // union  
  
never // unreachable  
  
enum Color {Red, Green, Blue = 4}  
let c: Color = Color.Green

## Declarations

let isDone: boolean;  
var isReady: boolean = false;  
  
function add(a: number, b: number): number {  
 return a + b;  
}

// **Return type is optional**

function subtract(a: number, b: number) {  
 return a - b;  
}

## Variable Assignment

// **Mutable**  
var x: number = 0;  
  
// **Imutable**  
let z = 0;

## Function Calls

helloWorld();  
function helloWorld(): string {  
 return 'Hello, world!';  
}  
  
let helloWorldAnonymous = (): string => {  
 return 'Hello, world!';  
};  
helloWorldAnonymous();

## Type Assertions

let numberString = '3';  
let numberLength: number = (numberString as string).length;

## Interfaces

// **Inline**

function printLabel(configuration: { label: string }) {  
 console.log(configuration.label);  
}  
printLabel({ label: 'This workshop will be in TypeScript' });

// **Explicit**

interface LabelOptions {  
 label: string;  
}

// **Optional Properties**

interface User {  
 name: string;  
 age?: number;  
}

// **Read Only**

interface UserReadOnly {  
 readonly name: string;  
}

## Type Aliases

type Name = string | string[];  
let aName: Name = 'aName';

## Function Types

interface FunctionalUser {  
 name: string;  
}  
let functionalUser = { name: 'atdd-tdd-typescript' };  
  
let printUserCallback = (user: FunctionalUser) => console.log(user.name);  
  
function getFunctionalUser(callback:   
 (user: FunctionalUser) => any) {  
 callback(functionalUser);  
}  
getFunctionalUser(printUserCallback);

## Classes

class Point {  
 x: number;  
 y: number;  
 static instances = 0;  
 constructor(x: number, y: number) {  
 this.x = x;  
 this.y = y;  
 }  
}  
let point = new Point(3, 4);

## Inheritance

class PointArea extends Point {  
 area(): number {  
 return x \* y;  
 }  
}  
  
interface Colored {  
 paint();  
}  
  
class Pixel extends Point implements Colored {  
 paint() {  
 // paint  
 }  
}

// **Short Fields Initialization**

class PointInitialization {  
 constructor(public x: number, public y: number) {}  
}  
let initializedPoint = new PointInitialization(1, 2);

## Generics

class Greeter<T> {  
 greeting: T;  
 constructor(message: T) {  
 this.greeting = message;  
 }  
}  
  
let greeter = new Greeter<string>('Hello ATDD/TDD, world!');

## Conditionals (if)

let value = 0;  
let message = '';  
  
if (value == 5) {  
 message = 'Success';  
} else if (value == 6) {  
 message = 'Acceptable';  
} else {  
 message = 'FAILURE';  
}

## Arrays

This is an array with nothing in it:

[]

This is an array with two numbers in it:

[1, 2]

This is an array with two numbers and a string in it. You can put anything into an array.

[1, 'hello!', 220]

Here's how you get something out of an array:

array = [1, 'hello', 220] array[0] # value is 1

Here's how you get the last element out:

array[-1] # value is 220

Here's another way to get the last element:

array.last # value is 220

Here's how you change an element:

array[0]= 'boo!' # value printed is 'boo!' # array is now ['boo', 'hello', 220]

How long is an array?

array.length # value is 3

Here's how you tack something onto the end of an array:

array.push('fred') # array is now ['boo', 'hello', 220, 'fred']

## Iteration

When you do something multiple times, it is called *iteration*. There are many ways to do this. The following will print *hello* five times:

5.times do

puts 'hello'

end

Here's one way to print the numbers from one to 10:

for x in 1..10 do

puts x

end

And here's another:

(1..10).each do |x|

puts x

end

The part between the *do* and the *end* is called a *block*. You can replace the *do* and *end* with braces:

(1..10).each { |x| puts x }

The *1..10* is a range, which works like an array of the numbers from 1 to 10. The *each* is a method that iterates through each element of the range. It is called an *iterator*.

This prints each value of an array:

["a", "b", "c"].each { |x| puts x }

What if you want to transform each element of an array? The following capitalizes each element of an array.

["hi", "there"].collect { |word| word.capitalize } # The result is ["Hi", "There"].

## Truth and Falsehood

If you try the examples above, you'll see that the ones that match print a number. That's the position of the first character in the match. The first expression (*/a/ =~ 'a string'*) returns *0*. (Ruby, like most programming languages, starts counting with 0.) The second returns *10*.

What happens if there's no match? Type this:

/^as/ =~ 'alas, no match'

and the result will be *nil*, signifying no match. You can use these results in an *if*, like this:

if /^as/ =~ some\_string puts 'the string begins with "as".' end

In Ruby, anything but the two special values *false* and *nil* are considered true for purposes of an *if* statement. So match results like *0* and *10* count as *true*.

## Blocks

A block is like a function without a name. It contains a set of parameters and one or more lines of code. Blocks are used a lot in Ruby. Iterators like *each* use blocks.

Here's how to search an array for an element:

gems = ['emerald', 'pearl', 'ruby'] gems.detect { |gem| /^r/ =~ gem } # returns "ruby"

The *detect* method takes a block as an argument. This block returns *true* if the first argument starts with an *r*. (Actually it returns *0*, which counts as *true*.) The *detect* method itself returns the first element for which its block is true.

When blocks are longer than one line, they are usually written using *do* and *end*. This is another way of writing the same code:

gems.detect do |gem| /^r/ =~ gem end

## Dictionaries

A *dictionary* lets you say "Give me the *value* corresponding to *key*." Dictionaries are also called *hashes* or *associative arrays*.

Here's how you create a dictionary:

dict = {}

Here's how you associate a value with a key:

dict['bret'] = 'texas' # looks a lot like an array, except that the key doesn't have to be a number.

Here's how you retrieve a value, given a key:

dict['bret'] # value is 'texas'.

Here's how you ask how many key/value pairs are in the dictionary:

dict.length # value is 1

What values does a dictionary have?

dict.values # value is the Array ['texas'].

What keys does it have?

dict.keys # value is the Array ['bret'].

## Expression Substitution

Expression substitution evaluates an expression within a string:

You can do this:

name = "Aidy"

puts "My name is: #{name}" => My name is: Aidy

And this:

puts "The sum of four times four is: #{4\*4}" => The sum of four times four is: 16

# WATiR Cheat Sheet

*NOTE: All specific single element selectors (ie. checkbox, table) also has a plural version that returns all instances of that element in the web page (ie. checkboxes, tables). You can apply filters to both the singular and plural version of the methods.*

## Browser

# start new driver session

browser = Watir::Browser.new :firefox

browser = Watir::Browser.new :chrome

browser = Watir::Browser.new :ie

# goto url

browser.goto "http://3Qilabs.com"

# refresh

browser.refresh

# close

browser.quit

## Textbox

# enter value

browser.text\_field(:id => "text").set "watir-webdriver"

# get value

browser.text\_field(:id => "text").value

# clear

browser.text\_field(:id => "text").clear

## Text Area

# enter value

browser.textarea(:id => "text").set "watir-webdriver"

# get value

browser.textarea(:id => "text").value

# clear

browser.textarea(:id => "text").clear

## Button

# is enabled?

browser.button(:id => "btn").enabled?

# button's text

browser.button(:id => "btn").text

# click

browser.button(:id => "btn").click

# find buttons with specific CSS class

browser.buttons(:class => "btn")[0].enabled?

## Checkbox

# check

browser.checkbox(:id => "btn").set

browser.checkbox(:id => "btn").set(true)

# uncheck

browser.checkbox(:id => "btn").clear

browser.checkbox(:id => "btn").set(false)

# is checked?

browser.checkbox(:id => "btn").set?

# click on 1st checkbox

browser.checkboxes[0].click

## Listbox

# select from list text

browser.select\_list(:id => "list").select "var"

# select using value

browser.select\_list(:id => "list").select\_value "var2"

# value is selected?

browser.select\_list(:id => "list").selected?("var2")

# get value

puts browser.select\_list(:id => "list").value

# get all items

browser.select\_list(:id => "list").options.each do |i|

puts "#{i.text}"

end

## Radio

# select value

browser.radio(:id => "radio").set

# is var selected?

browser.radio(:id => "radio").set?

## Image

# is image loaded?

browser.image(:src => "img.gif").loaded?

# height

browser.image(:src => "img.gif").height

# width

browser.image(:src => "img.gif").width

# click

browser.image(:src => "img.gif").click

# click 1st image

browser.images[0].click

## Div

# get text

browser.div(:class => "body").text

# get text of 2nd div when it appears

browser.divs[1].when\_present.text

## Table

# row 1, col 1

browser.table(:id => "table")[0][0].text

# row 1, col 2 (alternate)

browser.table(:id => "table").tr{0}.cell{1}.text

# row 2 - entire text

puts browser.table(:id => "table")[1].text

# click row #4

puts browser.table(:id => "table")[3].click

# get column count

browser.table(:id => "table").row.cells.length

# row count

browser.table(:id => "table").row\_count

browser.table(:id => "table").rows.length

## General

# [exists?]

browser.text\_field(:id => "text").exists?

# [enabled?]

browser.select\_list(:id => "list").enabled?

# [present?]

browser.element(:id => "e").present?

# [tag\_name]

browser.element(:id => "e").tag\_name

# [screenshot]

browser.screenshot.save "c:\\page.png"

# [to\_subtype] # returns button

browser.element(:id => "btn").to\_subtype

# [index] click 2nd image on page

browser.image(:index => 1).click

# [loops]

# get names of all text-fields

browser.text\_fields.each do |i|

puts i.name

end

# get name of first text-field

puts browser.text\_fields[0].name

# get name of second text-field

puts browser.text\_fields[1].name

# html text of the page

browser.html

# html text of a table

browser.table(:id => "table").html

## Waiting

# [wait\_until\_present]

browser.button(:id => "btn").when\_until\_present

# [when\_present]

browser.button(:id => "btn").when\_present.click

browser.button(:id => "btn").when\_present(10).click

# [wait\_while\_present]

browser.button(:value => "submit").click

browser.button(:value => "submit").wait\_while\_present

# [implicit wait] 5 seconds

# good to have, but don't recommend its global use

browser.driver.manage.timeouts.implicit\_wait = 5

# PageObject Cheat Sheet

*NOTE: The complete PageObject gem documentation is available at:* [*https://github.com/cheezy/page-object/wiki*](https://github.com/cheezy/page-object/wiki) *A subset is presented here.*

## Create your page

The first thing you must do is create your pages. These are simple ruby classes that include the PageObject Module. Do not create your own initialize method as one already exists and should not be overwritten. If you do need to complete some page initialization then you could implement a initialize\_page callback method which will be called after the page is created and all dynamic methods are added.

class RegistrationPage

include PageObject

page\_url "http://mysite.com/registration"

...

end

By including the PageObject Module you have added a lot of capabilities to your page. Let's take a look at how we might use some of that right now.

## Describe your page

After you create your class you need to describe the web page this class represents. The RegistrationPage example might look like this:

class RegistrationPage

include PageObject

page\_url "http://mysite.com/registration"

text\_field(:name, :id => 'name')

text\_field(:email, :id => 'email')

button(:register, :value => 'Register')

end

By calling these methods, the PageObject module will add several additional methods for you. To learn about what methods are available and the generated methods please see [the Accessors page](http://rubydoc.info/github/cheezy/page-object/master/PageObject/Accessors).

## Use your page

Now that we have a basic page object defined it is time to put it to use. You can use either watir-webdriver or selenium-webdriver as the driver gem. Just pass them into the constructor.

browser = Watir::Browser.new :chrome registration\_page = RegistrationPage.new(browser)

or

browser = Selenium::WebDriver.for :chrome registration\_page = RegistrationPage.new(browser)

Once created, you can interact with the page using the generated methods.

registration\_page.name = 'Test User' registration\_page.email = 'test@example.com' registration\_page.register

That is all there is to getting started with this gem. The good news is that there is a lot of additional functionality that was not covered in this brief introduction.

# Cucumber Cheat Sheet

*The Cucumber documentation is very extensive. You can view it online at:* [*https://github.com/cucumber/cucumber/wiki/A-Table-Of-Content*](https://github.com/cucumber/cucumber/wiki/A-Table-Of-Content) *A subset is presented here.*

## Scenario

A scenario is a concrete example that illustrates a business rule. It consists of a list of steps.

You can have as many steps as you like, but we recommend you keep the number at 3-5 per scenario. If they become longer than that they lose their expressive power as specification and documentation.

In addition to being a specification and documentation, a scenario is also a test. As a whole, your scenarios are an executable specification of the system.

Scenarios follow the same pattern:

* Describe an initial context
* Describe an event
* Describe an expected outcome

This is done with steps.

## Steps

A step typically starts with Given, When or Then. If there are multiple Given or When steps underneath each other, you can use And or But. Cucumber does not differentiate between the keywords, but choosing the right one is important for the readability of the scenario as a whole.

### Given

Given steps are used to describe the initial context of the system---the scene of the scenario. It is typically something that happened in the past.

When Cucumber executes a Given step it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to the test database.

It's ok to have several Given steps (just use And or But for number 2 and upwards to make it more readable).

### When

When steps are used to describe an event, or an action. This can be a person interacting with the system, or it can be an event triggered by another system.

It's strongly recommended you only have a single When step per scenario. If you feel compelled to add more it's usually a sign that you should split the scenario up in multiple scenarios.

### Then

Then steps are used to describe an expected outcome, or result.

The step definition of a Then step should use an assertion to compare the actual outcome (what the system actually does) to the expected outcome (what the step says the system is supposed to do).

## Step Arguments

A scenario contains multiple Given/When/Then steps. An example of a step that can have multiple values in between the double-quotes can be used to provide values to its Step Definition.

When I click on “Adopt Me”

Translates into a Step Definition written in Ruby:

When(/^I click on "([^"]\*)"$/) do |arg1|

# Write code here that turns the phrase above into concrete actions

end

**Arg1** will contain the value of the string that was in between the double-quotes based upon the Regular Expression in the Step Definition.

## Data Tables

A Data Table is handy for passing a list of values to a step definition:

When I fill the form in with the following values:

| element | value |

| order\_name | Joe Sixpack |

| order\_address | 123 Main St. |

| order\_email | joe@sixpack.com |

| order\_pay\_type | Check |

This translates into a Step Definition written in Ruby:

When(/^I fill the form in with the following values:$/) do |table|

# table is a Cucumber::MultilineArgument::DataTable

end

When reviewing the Cucumber documentation, the table is passed in as a Cucumber object that is essentially an array of hashes. You can then iterate over the data that has been passed in and do something within your Step Definition.

## Scenario Outline

A Scenario Outline allows you to define a scenario that can be repeated for every row in the Examples table that follows the outline. A Scenario Outline looks like this:

Scenario Outline: All fields are required to complete an adoption

Given I have added a puppy to my litter

When I tap "Complete the Adoption"

And everything is filled in

And I yet leave <element> blank

And I click "Place Order"

Then I see "<error>"

Examples:

| element | error |

| order\_name | Name can't be blank |

| order\_address | Address can't be blank |

| order\_email | Email can't be blank |

| order\_pay\_type | Pay type can't be blank |

Notice the words in the **<>** brackets? These correspond to the columns of data in the Examples table. For each row past the header, the scenario will repeat and plug in the values from the table in the appropriate spots within the outline.

## Tags

Tags are a way to group Scenarios. They are @-prefixed strings and you can place as many tags as you like above Feature, Scenario, Scenario Outline or Examples keywords. Space character are invalid in tags and may separate them.

Tags are inherited from parent elements. For example, if you place a tag above a Feature, all scenarios in that feature will get that tag.

Similarly, if you place a tag above a Scenario Outline or Examples keyword, all scenarios derived from examples rows will inherit the tags.

You can tell Cucumber to only run scenarios with certain tags, or to exclude scenarios with certain tags.

Cucumber can perform different operations before and after each scenario based on what tags are present on a scenario.

An example of a tagged scenario:

@wip

Scenario: Be able to view the details of a puppy

Given I am on the home page

When I click on the first puppy

Then I see "Adopt Me!"

## Reports

Cucumber can report results in several different formats, using formatter plugins. The available formatters plugins are:

* Pretty
* HTML
* JSON
* Progress
* Usage
* JUnit
* Rerun

Note that some Cucumber implementations might not provide all of these formatter plugins, and some implementations might provide additional ones.

## Text

Text can be embedded into the report from both Step Definitions and Hooks:

When /I do something pertinent/ do

puts ‘This goes into the report’

Kernel.puts ‘This goes to STDOUT, but not into the report’

end

## Calling a step or steps from within a step

### Calling steps

You can call a step from within another step with the following syntax.

Given(/^I am logged in$/) do

step “I am on the home page”

step ‘I enter “john@doe.com” into the “username” field’

step ‘I enter “Passw0rd1” into the “password” field’

end

### Calling steps that have a Data Table as an argument

You can use a Ruby string literal syntax to pass data to a scenario that requires a data table.

steps %Q{

And I fill the form in with the following values:

| element | value |

| name | Joe Sixpack |

| address | 123 Main St. |

| email | joe@sixpack.com |

| pay\_type | Check |

}

## Cucumber Configuration

Cucumber lets you store and reuse commonly used cucumber command line arguments for a project in a cucumber.yml or cucumber.yaml file

### Defining Profiles

# cucumber.yml

##YAML Template

---

html\_report: --format progress --format html --out=features\_report.html

bvt: --tags @bvt

Defining a template requires a name and then the command-line options that you want to execute with this profile. The example above generates two profiles: the first, named html\_report, with a list of command-line options that specify new output formats and a second, named bvt which executes all features and scenarios tagged with @bvt.

### Executing Profiles

[user@system project] cucumber --profile html\_report

[user@system project] cucumber -p bvt

The execution of a profile simply requires the use of the flag --profile or -p.

During execution you can also specify additional parameters alongside the profile.

[user@system project] cucumber --profile html\_report --tags ~@wip

Even multiple profiles can be specified together. The following executes all the features and scenarios tagged as @bvt with the specified progress and html output.

[user@system project] cucumber -p html\_report -p bvt

### Default Profile

It is often the case that you will want to execute Cucumber with a particular profile a majority of the time. The Cucumber configuration file uses a default profile to provide this functionality. When you specify a default profile you are stating that Cucumber should apply this command-line options to an execution when you do not specify a profile.

Using the same example, perhaps we want the html\_report profile to be our default execution.

# cucumber.yml

##YAML Template

---

default: --profile html\_report --profile bvt

html\_report: --format progress --format html --out=features\_report.html

bvt: --tags @bvt

The default profile is a special profile that when present, is applied to the execution of Cucumber when you have not specified a profile.

[user@system project] cucumber

So now, by default, Cucumber is going to use both the bvt profile and html\_report profile testing all features and scenarios tagged as @bvt with the progress output and html output.

# RSpec Cheat Sheet

RSpec is a testing tool for Ruby, created for behavior-driven development (BDD). RSpec has many useful capabilities to use for testing, but we will specifically use the Expectations support in the class.

## 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

Does the object have these methods?

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 "oh" }.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)

# Appendix

TypeScript Cheat Sheet – <https://devhints.io/typescript>

WATiR Cheat Sheet - <https://awetest.zendesk.com/hc/en-us/articles/201883796-Watir-Webdriver-Cheatsheet>

PageObject Cheat Sheet – <https://github.com/cheezy/page-object/wiki/Get-me-started-right-now!>

Cucumber Wiki - <https://github.com/cucumber/cucumber/wiki/A-Table-Of-Content>

RSpec Cheat Sheet - <https://www.anchor.com.au/wp-content/uploads/rspec_cheatsheet_attributed.pdf>