# 4. Getting started with Cucumber

Our first Cucumber project

We are finally ready to add Cucumber to our testing toolbox. The downloaded source has a project we will start with. The project is in the `learn_cucumber` directory. Let's spend a few minutes and look at the directory structure and a couple of files. If you are using *RubyMine* go ahead and open the directory. Otherwise open the directory with a tool that shows the directories and files.
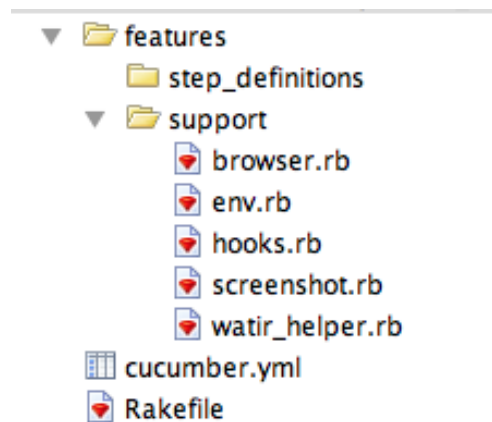
The first thing you will notice about our cucumber project is that there is a single directory at the root of the project named `features`. This directory is where all of your cucumber features will reside. In this directory you will find two additional directories. They are `support` and `step_definitions`. The `step_definitions` directory is empty at this time. The `support` directory has several files. Let's look at a few of them.

In the `support` directory open the `browser.rb` file. This file is responsible for making the decision of which browser our tests will use. The default browser on the Windows platform is Internet Explorer, Safari on the OS X platform, and Celerity on Jruby. In each case you can override the default browser by setting an environment variable named `FIREWATIR`. When the variable is set cucumber will use Firefox.

> **Note => JRuby - the Java Platform**
> JRuby is a Java Implementation of the Ruby Programming Language. It is know to be fast and stable. <MORE HERE>

Another file in the *support* directory is `screenshot.rb`. This file will be used to take a snapshot of the browser when a test fails. The final file we will look at for now is `hooks.rb`.

```
Before do
  @browser = Browser.new
end

After do
  @browser.close
end

After do | scenario |
  embed_screenshot("sshot-#{Time.new.to_i}") if scenario.failed?
end
```

Here you can see that we have a `Before` block and two `After` blocks. The `Before` block is opening up the browser and runs before each test. The first `After` block is closing the browser and runs after each test. The second `After` block is calling a method that will take a snapshot of the browser but only if the scenario fails.

Writing a simple cuke

Let's write our first cucumber script. Create a file in the `features` directory called `making_cheese.feature`. We will start with our story description.

```
Feature: Making Cheese

  As a cheese maker
  I want to make cheese
  So I can share my cheesiness
```

This seems simple enough. At this point we are providing documentation to the readers that explains the purpose of this feature we are building. This is truly free-form text except for the first word - *Feature*. Each feature file must have one *Feature* description. The colon after the *Feature* keyword is important - `cucumber` will complain without it. All of the documentation you add will be contained in the reports that are produced when we run the `cucumber` command.

The remainder of the file is filled with *Scenarios*. A *Scenario* is nothing more than an example of how the user will interact with the system. The combination of all of the scenarios define the story. Let's look at one.

```
  Scenario: Using the cheese machine
    Given I have no cheese
    When I press the make cheese button
```

```
    Then I should have 1 piece of cheese
```

There are four keywords in this scenario. The first is *Scenario*. As we said before, a *Scenario* describes how the user will interact with the system. Cucumber supports several types of *Scenarios*. The example above is the simplest form but we will look at another type later in this chapter. The description for the scenario should describe what is unique about this particular *Scenario* and the keyword must end with a colon.

After the *Scenario* we have what cucumber calls steps. These steps begin with one of four keywords - *Given, When, Then,* and *And*. This format is known as *Gherkin*. *Gherkin* is a Business Readable Domain Specific Language that describes a software's behavior without defining how the behavior is implemented. It is quickly becoming an industry standard and is implemented by numerous tools.

Given steps are used to define context and perform setup necessary for the *Scenario*. *When* steps define the action that is taken as a part of the *Scenario*. *Then* is where you verify the expected results. *Given, When,* and *Then* can have additional steps that begin with *And*. This will become clearer as we write more *Scenarios*.

It's time to run the script and see what happens. If you are using *RubyMine* right-click on the filename and select the run menu option. If you are running from command-line type `cucumber feature\making_cheese.feature`. You should see output that look similar to the following.

```
Feature: Making Cheese
  As a cheese maker
  I want to make cheese
  So I can share my cheesiness

  Scenario: Using the cheese machine
    Given I have no cheese
    When I press the make cheese button
    Then I should have 1 piece of cheese

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.232s


You can implement step definitions for undefined steps with
these snippets:


Given /^I have no cheese$/ do
  pending # express the regexp above with the code you wish …
end


When /^I press the make cheese button$/ do
```

```
  pending # express the regexp above with the code you wish …
end


Then /^I should have (\d+) piece of cheese$/ do |arg1|
  pending # express the regexp above with the code you wish …
end
```

Cucumber is informing us that there are no step definitions for the scenario. It is also providing a starting point for you to implement those steps. Let's create a new file in the step_definitions directory named making_cheese_steps.rb. Copy the step definitions provided by cucumber to that file. When you run the cucumber command again you should see something slightly different.

```
Feature: Making Cheese
  As a cheese maker
  I want to make cheese
  So I can share my cheesiness

  Scenario: Using the cheese machine
    Given I have no cheese
      TODO   (Cucumber::Pending)
      ./features/step_definitions/making_cheese_steps.rb:3:in …
    When I press the make cheese button
    Then I should have 1 piece of cheese

1 scenario (1 pending)
3 steps (2 skipped, 1 pending)
```

Here you can see that it found the steps but indicated that the first step was pending. As a result it skipped the remaining steps. When cucumber runs into a step that is not defined it does not continue running the scenario. We can easily resolve this. Let's write some code to make the steps run.

```
Given /^I have no cheese$/ do
  puts "I am so sad.  I have no cheese"
end

When /^I press the make cheese button$/ do
  puts "Yeah.  I have a cheese machine"
end

Then /^I should have (\d+) piece of cheese$/ do |num_pieces|
  puts "I now have #{num_pieces} pieces of cheese."
end
```

Notice that I have replaced the arg1 value in the third step definition with

`num_pieces`. This is to add clarity. Let's run the scenario by executing `cucumber feature\making_cheese.feature` from the command-line again. This time you will notice that all of the scenarios pass and the messages were printed to the screen.

You might be asking yourself why this scenario is passing. We haven't actually done anything yet so why should it work? Well cucumber is following the same pattern we saw when we wrote the *Watir* scripts in the last chapter. It is assuming success until either it encounters a code error or an assertion fails. We will see how this all comes together in the next section.

Purchase a book with cucumber

Now it is time to write a cucumber feature that does something more interesting than the feature we created in the last section. In this section we will write a feature that purchases a book. Our feature file should begin like this:

```
Feature: Purchasing books
  As an online shopper
  I want to purchase books on the Internet
  So I do not have to make a trip to the book store
```

This is a good start. Now we need to begin writing the first *Scenario*. Let's start with this:

```
  Scenario: purchase one book
    Given I am on the depot site
    When I press the Add to Cart button
    And I select the Checkout link
    And I enter "Cheezy" in the name field
    And I enter "123 Main Street" in the address field
    And I enter "cheezy@example.com" in the email field
    And I select "Credit card" from the pay with dropdown
    And I press the Place Order button
    Then I should see "Thank you for your order"
```

Now go ahead and generate the step definitions and rename the variables to something more meaningful. Place the step definitions in a file named `purchase_books_steps.rb` in the `step_definitions` directory. After the step definitions are generated I would like you to try to complete the script to make the *Scenario* run successfully. Here is the completed first step:

```
Given /^I am on the depot site$/ do
  @browser.goto "http://localhost:3000/store"
end
```

Try to complete all of the remaining steps without referring to the work we did in the last

chapter. Repetition will help us learn the *Watir* syntax and make us better automated testers. When you are finished go ahead and run the *Feature*. If you have a failure, correct it and try again until the script runs all of the way through.

RSpec Matchers

In the last chapter we verified the success message with the following code:

```
fail unless browser.text.include? "Thank you for your order"
```

At this time I would like to explore an alternative to this approach. Our cucumber project already includes the *require* statements to include a rubygem named *RSpec*. This gem will help us rewrite this step to make it cleaner and make our error message better.

The RSpec gem includes something called matchers. This is simply a way to perform some form of verification. In the appendix *RSpec Matchers* you can find a nearly complete listing of all of the matchers available to you but we will cover the most simple form here.

First of all, RSpec adds the `should` and `should_not` methods to all objects. This allows us to specify our expectations in a near english form. For example we can say that some object `should` or `should_not` do something. In our case we want to say that the browser should include some text. We do this by calling the `include?` method on the text returned from the browser object.

Secondly, RSpec allows us to drop the question mark in method calls. This is to allow for more expressive expectations. For example, we can write our step definition like this:

```
Then /^I should see "([^\"]*)"$/ do |expected|
  @browser.text.should include expected
end
```

This literally calls the `include?` method on `@browser.text` and passes the string `"Thank you for your order"`. If the method call returns true it passes. If it returns false it fails.

We will introduce more RSpec matchers as we move through the examples.

Your finished step definition file should look like this:

```
Given /^I am on the depot site$/ do
  @browser.goto "http://localhost:3000/store"
end

When /^I press the Add to Cart button$/ do
```

```ruby
  @browser.button(:value => "Add to Cart").click
end

When /^I select the Checkout link$/ do
  @browser.link(:text => "Checkout").click
end

When /^I enter "([^\"]*)" in the name field$/ do |name|
  @browser.text_field(:id => "order_name").set(name)
end

When /^I enter "([^\"]*)" in the address field$/ do |address|
  @browser.text_field(:id => "order_address").set(address)
end

When /^I enter "([^\"]*)" in the email field$/ do |email|
  @browser.text_field(:id => "order_email").set(email)
end

When /^I select "([^\"]*)" from the pay with dropdown$/ do |pay_type|
  @browser.select_list(:id => "order_pay_type").select(pay_type)
end

When /^I press the Place Order button$/ do
  @browser.button(:value => "Place Order").click
end

Then /^I should see "([^\"]*)"$/ do |expected|
  @browser.text.should include expected
end
```

Purchase two books

Let's write a second *Scenario* in the same feature file called "Purchase two books" and place it in the same file below the previous *Scenario*. As you might guess, we want this new *Scenario* to purchase two books and complete the order. Your *Scenario* should be nearly identical to the previous *Scenario* with the addition of two new steps. You will need to generate a step definition for one of those steps. Here are the new steps:

```
    And I select the Continue shopping link
    And I press the Add to Cart button
```

Go ahead and generate the step definition and implement the method. As you see, we were able to reuse all of the previous step definitions in this new *Scenario* and really only had to add one line of code to make the new *Scenario* work.

Scenario Outlines

Cucumber has another type of *Scenario* called a *Scenario Outline*. It allows us to separate out the data used in the *Scenario* from the description of the steps in the *Scenario* and thereby run it with multiple sets of data. This is very useful for situations when you have a single path through the application that you wish to run with different data.

Scenario Outlines take this form:

```
Scenario Outline: Using the cheese machine
  Given I have no Cheese
  When I press the make "<type>" cheese button
  Then I should see the "<message>" message

Scenarios:
  | type      | message               |
  | Swiss     | I love Swiss cheese   |
  | Blue      | I love Blue cheese    |
  | Cheddar   | I love Cheddar cheese |
```

As you can see, *Scenario Outlines* have two parts. The first is the outline which provides the steps for the test and the second is the data used in the steps. These two parts are bound together by variables that are replaced when the *Scenario* runs. The variables are defined in the steps with angle brackets like <name>. This is matched with a heading in the *Scenarios* section and the value for each row is inserted into the step.

The *Scenario* above will run three times. The first time the *type* variable will be replaced with the word *Swiss* and the *message* variable will be replace with *I love Swiss cheese*. The second time the *Scenario* is run it will use the data from the second row and so on.

For you next assignment I want you to rewrite the first *Scenario* to make it a *Scenario Outline* and provide three *Scenarios* so it will run with three complete sets of data. The *Scenarios* portion of your script will look something like:

```
Scenarios:
  | name    | address     | email             | pay_type    |
  | Cheezy  | 123 Main St | cheezy@example.com| Credit card |
```

```
| Joseph | 555 South St| joe@guru.com       | Check        |
| Jared  | 234 Leandog | doc@dev.com        | Purchase order|
```

Background

Our *Scenarios* have some duplication that we want to address. Both of them begin with the same Step:

```
    Given I am on the depot site
```

Since we have learned that duplication is evil we should find a way to eliminate this travesty. Luckily `cucumber` has a solution. It is call *Background*. Think of it as a step or collection of steps that are executed prior to each *Scenario*. Using a *Background* the duplication in our *Scenarios* is eliminated and we end up with:

```
  Background:
    Given I am on the depot site

  Scenario Outline: purchase one book
    When I press the Add to Cart button
    …

  Scenario: purchase two books
    When I press the Add to Cart button
    …
```

Now that feels better, doesn't it.

Verify the shopping cart

We are off to a good start in cucumber land but so far our tests haven't really tested much. We are about to change that. In this section we will write some *Scenarios* that will test our shopping cart page. Let's look at a shopping cart.

Looking at this shopping cart what *Scenarios* do you think we need to verify correctness?

I can think of three *Scenarios* I would want to write to verify this shopping cart. They are

1. Verify the shopping cart with one book.
2. Verify the shopping cart with two of the same books.
3. Verify the shopping cart with two different books.

These are the three scenarios I want you to write now. Each time I want you to verify the values in the "Qty", "Description", "Each" and "Total" columns as well as the shopping cart "Total".

The shopping cart data is stored in a html **table**. You can access the fields in a table by using rows and columns. The *Watir Quick Reference* provides an example of how to do this. The shopping cart "Total" can be accessed directly using the `cell` method. Be aware! This is by far the most complex exercise we have done so far and I expect it to take a fair amount of time to complete.

Try to complete this exercise on your own before looking at the completed scripts below.

The completed scripts
Here are the *Scenarios* I came up with for this exercise:

```
  Scenario: Verify shopping cart for a single book
    When I add "Pragmatic Project Automation"
    Then I should see "1" in the quantity
    And I should see "Pragmatic Project Automation" in the
description
    And I should see "29.95" in the each
    And I should see "29.95" in the subtotal
    And I should see "29.95" for the cart total

  Scenario: Verify shopping cart for two of the same book
    When I add "Pragmatic Project Automation"
    And I select the Continue shopping link
```

```
    And I add "Pragmatic Project Automation"
    Then I should see "2" in the quantity
    And I should see "Pragmatic Project Automation" in the
description
    And I should see "29.95" in the each
    And I should see "59.90" in the subtotal
    And I should see "59.90" for the cart total

  Scenario: Verify shopping cart for two different books
    When I add "Pragmatic Project Automation"
    And I select the Continue shopping link
    And I add "Pragmatic Version Control"
    Then I should see "1" in the quantity for item "1"
    And I should see "Pragmatic Project Automation" in the
description for item "1"
    And I should see "29.95" in the each for item "1"
    And I should see "29.95" in the subtotal for item "1"
    And I should see "1" in the quantity for item "2"
    And I should see "Pragmatic Version Control" in the
description for item "2"
    And I should see "28.50" in the each for item "2"
    And I should see "28.50" in the subtotal for item "2"
    And I should see "58.45" for the cart total
```

I could have rewritten the first two *Scenarios* to use the form where I provide the line item which is used in the third but I thought it was easier to understand in the form above.

Here are the step definitions for the *Scenarios* above.

```
When /^I add "([^\"]*)"$/ do |book_name|
  book = 0
  book = 1 if book_name == "Pragmatic Project Automation"
  book = 2 if book_name == "Pragmatic Unit Testing (C#)"
  book = 3 if book_name == "Pragmatic Version Control"
  @browser.button(:value => "Add to Cart", :index => book).click
end

def line_item(line)
  @browser.table(:index => 1)[line+2]
end

Then /^I should see "([^\"]*)" in the quantity$/ do |quantity|
  line_item(1)[1].text.should include quantity
end
```

```
Then /^I should see "([^\"]*)" in the description$/ do |
description|
  line_item(1)[2].text.should == description
end

Then /^I should see "([^\"]*)" in the each$/ do |each|
  line_item(1)[3].text.should == "$#{each}"
end

Then /^I should see "([^\"]*)" in the subtotal$/ do |subtotal|
  line_item(1)[4].text.should == "$#{subtotal}"
end

Then /^I should see "([^\"]*)" for the cart total$/ do |total|
  @browser.cell(:class => "total-cell").text.should == "$#
{total}"
end

Then /^I should see "([^\"]*)" in the quantity for item "([^\"]
*)"$/ do |quantity, line|
  line_item(line.to_i)[1].text.should include quantity
end

Then /^I should see "([^\"]*)" in the description for item "([^
\"]*)"$/ do |description, line|
  line_item(line.to_i)[2].text.should == description
end

Then /^I should see "([^\"]*)" in the each for item "([^\"]*)"$/
do |each, line|
  line_item(line.to_i)[3].text.should == "$#{each}"
end

Then /^I should see "([^\"]*)" in the subtotal for item "([^\"]
*)"$/ do |subtotal, line|
  line_item(line.to_i)[4].text.should == "$#{subtotal}"
end
```

There are many new things to discuss there. The first is the way I select the `index` in the call to add a book to my cart.

```
When /^I add "([^\"]*)"$/ do |book_name|
  book = 0
  book = 1 if book_name == "Pragmatic Project Automation"
  book = 2 if book_name == "Pragmatic Unit Testing (C#)"
  book = 3 if book_name == "Pragmatic Version Control"
```

```
@browser.button(:value => "Add to Cart", :index => book).click
end
```

Here I am creating a local variable named `book` and setting its' value to `0`. The next thing I am doing is comparing the `book_name` parameter to the know book names and setting the `book` variable to the appropriate index. This is a very simple, although brittle, way of taking the name of a book and making sure we click the correct "Add to Cart" button.

The next thing to explore is the method we added to remove the duplication of accessing the table.

```
def line_item(line)
  @browser.table(:index => 1)[line+2]
end
```

The only way to find the table is to access it by index. You will note that we are using the `[]` notation. In our case we are using the value passed in and adding `2`. This is to account for the two line header that exists on the table.

Next we will explore the call that uses this method.

```
line_item(1)[1].text.should include quantity
```

In this line we are passing a `1` since this step always wants access to the first line item. We are also accessing the first column by using the `[1]` notation. Again, the quantity is in the first column and this call returns the text in that column. Since the quantity column has another character in it we use the `include` method in the comparison.

If you look at how we compared the subtotal you will see this line:

```
line_item(1)[4].text.should == "$#{subtotal}"
```

Instead of using `include` we are using `==` because we expect an exact match. Also, we are building up a string to match that includes the dollar sing since we didn't include the dollar sign in the *Scenario*.

The final thing I wish to look at is this step definition:

```
Then /^I should see "([^\"]*)" in the description for item "([^
\"]*)"$/ do |description, line|
  line_item(line.to_i)[2].text.should == description
end
```

Notice that we are passing in the `line` value to the step. Cucumber is passing it as a

string and yet our method is expecting it to be a number. We can convert it from a string to a number by using the `to_i` method.

Introducing Classes

In chapter three we introduced the concept of a `module`. We learned it is a place to store methods we wish to use in different places. In this section we want to introduce `classes`. They are very similar to `modules`. The main difference we wish to explore here is that a `class` can have data in addition to methods. `modules` can be inserted into `classes` as well.

A class is defined in much the same way as modules.

```
class MyClass
   …
end
```

Inside the class you can put methods and instance variables. In this chapter we will be building classes to represent pages on our website.

A page object for our shopping cart

The step definitions we wrote for verifying the shopping cart have a major problem. They are extremely brittle. What I mean by that is that many steps will break if the page is changed. If a designer changes the order of the columns in the table this will break the three latest *Scenarios.* I will have to search through all of the step definitions to find all of the changes necessary to fix the broken tests. If a designer decides to change the page to use *div*s instead of the *table* then will have to do the same.

Brittle tests are a major cause of many test automation failures. Tests that fail every time the application changes and that need significant effort to fix lead us to spend more and more of our time keeping existing tests running. I have seen companies where nearly a third of the testers time is spent keeping tests running and that takes away from writing new tests and activities outside of automation like exploratory testing.

But what can we do? It is fine for multiple tests to break when the application changes but it is not fine for us to have to go the multiple places to fix it. When the tests fail we really want to go to **one** well know place to make the appropriate changes and then see all tests pass.

Let's introduce a *Page Object* to hide all of the details about our shopping cart page. A *Page Object* is a *class* that contains all of the code to access the web page. It should know the details of this page and should be used to protect us from change to this page. In other words, if we are successful in this effort we should not have to go to any other place to make changes if the web page changes. Let's create the basic structure first

and then we will fill in the details later.

```
class ShoppingCartPage

  def initialize(browser)
    @browser = browser
  end

end
```

To begin with we have added an `initialize` method. In our page we will need to have access to the `@browser` variable and since it lives outside of our class we need to pass it in. The `initialize` method is call when we create a new object of this class via the `initialize` method and all parameters are pass through. For example, the following code will in turn call our initialize method.

```
@shopping_cart = ShoppingCartPage.new(@browser)
```

Now that we have the ability to create our page object we need to add some behavior. Let's start by moving the behavior to get the value of the "quantity" and "description" columns into our class.

```
  def quantity_for_line_item(line_number)
    line_item(line_number)[1].text
  end

  def description_for_line_item(line_number)
    line_item(line_number)[2].text
  end

  def line_item(line)
    @browser.table(:index => 1)[line+2]
  end
```

Using this pattern I complete all of the methods that provide the details of the shopping cart page. When I am done I update my step definitions. First of all I need to determine where I must create my *Page Object* and next I must replace the steps with calls to that object. Here is some of the code.

```
When /^I add "([^\"]*)"$/ do |book_name|
  book = 0
  book = 1 if book_name == "Pragmatic Project Automation"
  book = 2 if book_name == "Pragmatic Unit Testing (C#)"
  book = 3 if book_name == "Pragmatic Version Control"
  @browser.button(:value => "Add to Cart", :index => book).click
  @cart = ShoppingCartPage.new(@browser)
```

```
end

Then /^I should see "([^\"]*)" in the quantity$/ do |quantity|
  @cart.quantity_for_line_item(1).should include quantity
end

…

Then /^I should see "([^\"]*)" in the quantity for item "([^\"]
*)"$/ do |quantity, line|
  @cart.quantity_for_line_item(line.to_i).should include
quantity
end

…
```

Go ahead and complete the modification of the other step definitions so that all calls for this page are delegated to the page object. When you are done your page object should look like this:

```
class ShoppingCartPage

  def initialize(browser)
    @browser = browser
  end

  def quantity_for_line_item(line_number)
    line_item(line_number)[1].text
  end

  def description_for_line_item(line_number)
    line_item(line_number)[2].text
  end

  def unit_price_for_line_item(line_number)
    line_item(line_number)[3].text
  end

  def subtotal_for_line_item(line_number)
    line_item(line_number)[4].text
  end

  def cart_total
    @browser.cell(:class => "total-cell").text
  end
```

```
  def line_item(line)
    @browser.table(:index => 1)[line+2]
  end
end
```

There are two additional things I would like to do to our *Page Object*.

Magic Numbers
Magic Numbers is a term used to describe numbers that appear in code that do not express their reason for existence or intent. An example of that would be the 2 in the following statement.

```
line_item(line_number)[2].text
```

I think we can make our code express our intent more clearly and simplify the maintenance of it by eliminating this magic. Let's introduce the idea of constants. A constant is a type of variable in Ruby that is supposed to remain constant. In other words, once it has a value it is not supposed to change. Constants are always capitalized in Ruby and take this form:

```
DESCRIPTION_COLUMN = 2
```

Placing this at the top of our class file allows us to replace the values like this:

```
line_item(line_number)[DESCRIPTION_COLUMN].text
```

Go ahead and replace all of the magic numbers throughout our class. Don't forget to replace the number that represents our header offset.

If the designer decides to change the order of the columns in the table we only have to change the numbers that are assigned to the constants representing the columns. If the designer decides to create an elaborate header that takes four rows we again only have to change the constant.

Keeping things private
All of the methods in our class were designed to be called by code that needs access to the shopping cart page. That is every method except line_item. This method was created to be used internally. Ruby has a private keyword which provides a way for us to make sure that internal methods stay internal. By placing our line_item method at the bottom of the class we can do the following:

```
  …

  private

  def line_item(line)
```

```
    @browser.table(:index => 1)[line+2]
  end
end
```

This ensures that this method will not be called by anyone except another method in this class.

Setting Values

Ruby allows us to add an equal sign to the end of methods. When we call these methods it looks like we are performing a simple assignment. Let's take a look at an example.

```
def address=(an_address)
  @browser.text_field(:id => "order_address").set(an_address)
end

# Can be called like this
@page.address = "123 Main Street"
```

Notice that there can be a space between the method name and the equal sign when we call the method. This makes the code easier to read and write. Let's see how we might use this in a *Page Object*.

Checking out

Armed with the ability to easily set values it is time to create a *Page Object* for the Checkout page. On this page we have a few text fields, a select list and a button. Let's give it a try.

```
class CheckoutPage

  def initialize(browser)
    @browser = browser
  end

  def name=(name)
    @browser.text_field(:id => "order_name").set(name)
  end

  def address=(address)
    @browser.text_field(:id => "order_address").set(address)
  end

  def email=(email)
```

```
      @browser.text_field(:id => "order_email").set(email)
   end

   def pay_with=(pay_with)
      @browser.select_list(:id => "order_pay_type").select
(pay_with)
   end

   def place_order
      @browser.button(:value => "Purchase Order").click
   end

end
```

Our goal is to make the step definitions as simple and intuitive as possible. Let's take a look:

```
When /^I select the Checkout link$/ do
   @browser.link(:text => "Checkout").click
   @checkout = CheckoutPage.new(@browser)
end

When /^I enter "([^\"]*)" in the name field$/ do |name|
   @checkout.name = name
end

When /^I enter "([^\"]*)" in the address field$/ do |address|
     @checkout.address = address
end

When /^I enter "([^\"]*)" in the email field$/ do |email|
   @checkout.email = email
end

When /^I select "([^\"]*)" from the pay with dropdown$/ do |
pay_type|
   @checkout.pay_with = pay_type
end

When /^I press the Place Order button$/ do
   @checkout.place_order
end
```

Notice that I created my page object in the first step definition. This is the point at which I am navigating to the checkout page in the browser. Again, this seems simple and intuitive but I think we make make it easier. Let see how in the next section.

WatirHelper

I have included a module in our project named `WatirHelper`. You will find it in the
`features/support` directory. This module is designed to make it easier to create
Page Objects. The way it works is you call some methods describing what things are
on our page and then it generates methods for you. Let's look at an example.

If we have a login screen with a username and password text field as well as a login
button we might create a class like this:

```ruby
class LoginPage
  include WatirHelper

  text_field(:username, :id => "user_id")
  text_field(:password, :id => "user_password")
  button(:login, :value => "Login")

  def initialize(browser)
    @browser = browser
  end
end
```

In this example we called a simple method for each element we wished to interact with.
Let's examine the first call in detail to see what it is doing. First of all we are passing
two arguments. The first parameter is the name we want to give the text field and the
second parameter is the locaters needed by *Watir* to find the element on the page.
When we call this method the `WatirHelper` module writes thee additional methods for
us.

```ruby
def username
  @browser.text_field(:id => "user_id").value
end

def username=(username)
  @browser.text_field(:id => "user_id").set(username)
end

def username_text_field
  @browser.text_field(:id => "user_id")
end
```

The first method returns the value contained within the text field. The second method
sets a value in the text field. The third method returns the text field. That is a lot of
code to be generated from one simple line. That is the beauty and power of
`WatirHelper`.

Let's make this a little more real. Below you see the `CheckoutPage` class rewritten using `WatirHelper`.

```
class CheckoutPage
  include WatirHelper

  text_field(:name, :id => "order_name")
  text_field(:address, :id => "order_address")
  text_field(:email, :id => "order_email")
  select_list(:pay_with, :id => "order_pay_type")
  button(:place_order, :value => "Place Order")

  def initialize(browser)
    @browser = browser
  end

end
```

This is much shorter and easier to maintain. As we said earlier, we always want to know where to go and what to change as the page evolves. Using `WatirHelper` helps define where we make changes. For example, if we assumed that something on the screen mockup was a button but it was later implemented as a link with an image we would only have to come to the top of the file and change the call to `button` with a call to `link`. If we discovered that a developer used a different `id` than what we had in the script we would simple change it at the top of the class. This makes it simpler and that is what we are striving for.

Since the new `CheckoutPage` was such a success let's see what happens when we introduce `WatirHelper` to our `ShoppingCartPage` object.

```
class ShoppingCartPage
  include WatirHelper

  link(:checkout, :text => "Checkout")
  link(:continue_shopping, :text => "Continue shopping")
  table(:cart, :index => 1)
  cell(:cart_total, :class => "total-cell")

  def initialize(browser)
    @browser = browser
  end

  def quantity_for_line_item(line_number)
    line_item(line_number)[1].text
```

```ruby
    end

  def description_for_line_item(line_number)
    line_item(line_number)[2].text
  end

  def unit_price_for_line_item(line_number)
    line_item(line_number)[3].text
  end

  def subtotal_for_line_item(line_number)
    line_item(line_number)[4].text
  end

  def line_item(line)
    cart[line+2]
  end
end
```

This didn't shrink the size of our class but it did clean it up a bit. This is due to the fact that most of our class already dealt with logic for retrieving specific data and not *Watir*. Notice I also added the two links on the page.

Convert all pages to use page objects
There is only one web page we have not created a *Page Object* for. That is the page that lists all of the books for purchase. Let's go ahead and change that over to a page object. After that we can change all of the step definitions to use our page objects.

```ruby
class CatalogPage
  include WatirHelper

  def initialize(browser)
    @browser = browser
  end

  def add_book_to_cart(name = "Pragmatic Project Automation")
    book = 0
    book = 1 if book_name == "Pragmatic Project Automation"
    book = 2 if book_name == "Pragmatic Unit Testing (C#)"
    book = 3 if book_name == "Pragmatic Version Control"
    @browser.button(:value => "Add to Cart",
        :index => book).click
  end
```

```ruby
  def visit
    @browser.goto "http://localhost:3000/store"
  end

end
```

This class is fairly small but we are introducing something completely new. The method to add a book to the cart has this strange syntax:

```ruby
  def add_book_to_cart(name = "Pragmatic Project Automation")
```

We are passing a variable and assigning a value to it at the same time. This is call a default value and it works like this. If you call the `add_book_to_cart` method passing in the string "Pragmatic Version Control" then the `name` variable will contain that value. If you call the `add_book_to_cart` method passing no value then "Pragmatic Project Automation" (the default value) is assigned to the `name` variable.

I think there is something we can do to clean up this method a little. But first let's talk about Ruby `Hashes`.

Hash
A `Hash` is a collection of name/value pairs. We have used them a lot already without pointing them out. Here are some examples of how you can use `Hashes`.

```ruby
# Create an empty hash
hs = {}

# Create a hash with three key/value pairs
hs = {1 => "a", 2 => "b"}

# Get the value associated with the key 1 and assign
# it to the val variable
val = hs[1]

# Put the value "z" in the hash associated with the key
# 1.  If that value already exists overwrite it.
hs[1] = "z"
```

Method cleanup
Using a `Hash` I think we can cleanup our method. Let's begin by defining a `Hash` to hold our book titles.

```ruby
BOOKS = {
  "Pragmatic Project Automation" => 1,
```

```
  "Pragmatic Unit Testing (C#)" => 2,
  "Pragmatic Version Control" => 3
}
```

In this example we simply mapped the title of the book to its' index on the page. There are definitely more robust ways to perform this but I we'll stick with this for now. The next step is to use this `Hash` in our method. Here is the updated version.

```
def add_book_to_cart(name = "Pragmatic Project Automation")
  @browser.button(:value => "Add to Cart",
      :index => BOOKS[name]).click
end
```

This is much simpler. We are just using the `name` as the key to look up the index in the `BOOKS Hash`.

Step Definitions
Now we can update all of our step definitions to use our *Page Objects*. I am including the entire listing here.

```
Given /^I am on the store page$/ do
  @catalog = CatalogPage.new(@browser)
  @catalog.visit
end

When /^I add a book to the cart$/ do
  @catalog.add_book_to_cart
  @shopping_cart = ShoppingCartPage.new(@browser)
end

And /^I checkout$/ do
  @shopping_cart.checkout
  @checkout = CheckoutPage.new(@browser)
end

And /^I continue shopping$/ do
  @shopping_cart.continue_shopping
end

And /^I enter "([^\"]*)" in the name field$/ do |name|
  @checkout.name = name
end

And /^I enter "([^\"]*)" in the address field$/ do |address|
  @checkout.address = address
end
```

```
And /^I enter "([^\"]*)" in the email field$/ do |email|
  @checkout.email = email
end

And /^I select "([^\"]*)" from the pay type dropdown$/ do |
pay_type|
  @checkout.pay_with = pay_type
end

And /^I place my order$/ do
  @checkout.place_order
end

Then /^I should see "([^\"]*)"$/ do |expected_text|
  @browser.text.should include expected_text
end

When /^I add "([^\"]*)" to the cart$/ do |book|
  @catalog.add_book_to_cart book
  @shopping_cart = ShoppingCartPage.new(@browser)
end

Then /^I should have the Pragmatic Project Automation book in
the cart$/ do
  @shopping_cart.description_for_line(1).should == "Pragmatic
Project Automation"
end

And /^I should have the Pragmatic Version Control book in the
cart$/ do
  @shopping_cart.description_for_line(2).should == "Pragmatic
Version Control"
end

And /^the quantity should be "(\d+)"$/ do |quantity|
  @shopping_cart.quantity_for_line(1).should include quantity
end

And /^the quantity should be "([^\"]*)" for line "([^\"]*)"$/ do
|quantity, line|
  @shopping_cart.quantity_for_line(line).should include quantity
end

And /^the unit price should be "([^\"]*)"$/ do |amount|
  @shopping_cart.unit_price_for_line(1).should == "$#{amount}"
```

```
end

And /^the unit price should be "([^\"]*)" for line "([^\"]*)"$/
do |amount, line|
  @shopping_cart.unit_price_for_line(line).should == "$#
{amount}"
end

And /^the subtotal should be "([^\"]*)"$/ do |amount|
  @shopping_cart.subtotal_for_line(1).should == "$#{amount}"
end

And /^the subtotal should be "([^\"]*)" for line "([^\"]*)"$/ do
|amount, line|
  @shopping_cart.subtotal_for_line(line).should == "$#{amount}"
end

And /^the cart total should be "([^\"]*)"$/ do |total|
  @shopping_cart.cart_total.should == "$#{total}"
end
```

Reusable partial pages

Many websites have a portion of a page duplicated on other pages. For example, it is quite common to see a common header and footer section on a site show up on most of the pages. It is also somewhat common to have a sidebar or menu show up on many pages.

Since we do not want to introduce duplication we need to find a way to script these common page elements once and use them in many pages. In Chapter 3 we introduced the concept of `modules` as a holder of methods we wish to use across multiple scripts. We can use the same idea here.

If you look at the google search page you will notice a set of links across the top of the page.

Web Images Videos Maps News Shopping Gmail more ▼

We can create a `module` to contain these links. Let's see what that would look like.

```ruby
module GoogleHeader
  include WatirHelper

  link(:web, :text => "Web")
  link(:images, :text => "Images")
  link(:videos, :text => "Videos")
  link(:maps, :text => "Maps")
  link(:news, :text => "News")
  link(:shopping, :text => "Shopping")
  link(:gmail, :text => "Gmail")

end
```

Now that we have the module let's see how we might use it in a google search page.

```ruby
class GoogleSearchPage
  include WatirHelper
  include GoogleHeader

  …
```

By simply including the module in our new page object we now have access to all of the links and we have eliminated the need to write the links in multiple places.

Refactor page objects to return page objects

You will notice on several of the step definitions we do something like click a link that takes us to a new page. The next line creates the page object that represents that page. In large projects it can be difficult to know where to create our new objects especially when you are reusing steps across multiple features. There is a simple solution to this problem. Let's explore it now.

Having our *Page Objects* return the next page is the simplest solution I have discovered. Let's see what this looks like. We can rewrite this step definition:

```
And /^I checkout$/ do
  @shopping_cart.checkout
  @checkout = CheckoutPage.new(@browser)
end
```

The new version will look like this:

```
And /^I checkout$/ do
  @checkout = @shopping_cart.goto_checkout_page
end
```

We will have to add a new method to our `ShoppingCartPage`.

```
def goto_checkout_page
  checkout
  CheckoutPage.new(@browser)
end
```

The first line of the method calls a method generated by `WatirHelper` and the second line creates and returns the new `CheckoutPage` object.

Go ahead and replace all such steps with this approach.

High level tests

Many of the *Scenarios* we have written so far suffer from a problem. They are too verbose. What I mean by this is they specify every single keystroke even when it is not important for the actual test. Let's take a look at the original *Scenario* that purchased a book.

```
  Scenario: purchase one book
    Given I am on the depot site
    When I press the Add to Cart button
    And I select the Checkout link
    And I enter "Cheezy" in the name field
    And I enter "123 Main Street" in the address field
    And I enter "cheezy@example.com" in the email field
    And I select "Check" from the pay with dropdown
    And I press the Place Order button
    Then I should see "Thank you for your order"
```

In this *Scenario* the only thing we are testing is the message received after completing an order. We can eliminate a lot of steps that really do not add to the clarity of the test by introducing a table. We have already seen tables in our *Scenario Outlines* but here we will add a table directly in our *Scenario*.

```
Scenario: purchase one book with a table
  When I press the Add to Cart button
  And I checkout with:
  | Name    | address         | email               | pay_type |
  | Cheezy | 123 Main Street | cheezy@example.com | Check    |
  Then I should see "Thank you for your order"
```

Let's see what the step definition looks like for this step:

```
And /^I checkout with$/ do |table|
  @checkout = @shopping_cart.goto_checkout_page
  @checkout.complete_order(table.hashes.first)
end
```

Notice that the step receives a variable named `table`. There is a `hashes` method on the table object. This method just returns an *Array* of *Hashes*. We haven't talked about *Arrays* yet so let's find out what they are.

Array
An array is a sequential ordered collection. This means that when you put an object in the first position of the array it is always in the first position unless you remove it. Let's look at how arrays can be used:

```
ar = []        # create an empty array
ar = [1,2,3]   # create an array with three elements
item1 = ar[0]  # place the item in the first position into item1
ar[1] = "foo"  # place the string "foo" in the second position
```

Finishing our high level test
The *Array* class has a `first` method which returns the first *Hash* in our table. This *Hash* uses the table headers as the key and the values in the table as the values.

Now we need to add one final method to our `CheckoutPage` to complete this effort. This method will take the *Hash* and use the `WatirHelper` generated methods to complete the order.

```
def complete_order(data)
  self.name = data['name']
  self.address = data['address']
  self.email = data['email']
  self.pay_with = data['pay_type']
  place_order
end
```

This is much nicer as the *Scenario* is at a higher level and as a result is not

overwhelming. But is it enough? I think we can go further and in the next section we'll see how.

Adding default data

It is fairly common for your *Scenarios* to require a lot of data in order to complete a successful run. In the last section we discussed how to use tables to create higher level tests. We should ask ourselves why do we need to provide all of this data if it is not important to what we are testing. Does providing this data really add clarity or is it distracting?

Our goal when writing *Scenarios* is to provide only the data that is necessary for the very specific thing we are testing. Everything else should be defaulted to some value that allows the *Scenario* to complete successfully. Let's start by creating a new *Scenario* that provides only the name and defaults everything else.

```
Scenario: purchase one book with partial default data
  When I press the Add to Cart button
  And I checkout with:
  | Name |
  | Bob  |
  Then I should see "Thank you for your order"
```

The step definition we wrote in the last section works with this code. It will pass a hash to the page object that contains only one entry - the *name* entry.

Let's see what default data looks like in our checkout page object.

```ruby
class CheckoutPage
  include WatirHelper

  DEFAULT_DATA = {
    "name" => "Cheezy",
    "address" => "123 Main Street",
    "email" => "cheezy@example.com",
    "pay_type" => "Credit card"
  }

  text_field(:name, :id => "order_name")
  text_field(:address, :id => "order_address")
  text_field(:email, :id => "order_email")
  select_list(:pay_with, :id => "order_pay_type")
  button(:place_order, :value => "Place Order")

  def initialize(browser)
```

```
    @browser = browser
  end

  def complete_order(data)
    data = DEFAULT_DATA.merge(data)
    self.name = data['name']
    self.address = data['address']
    self.email = data['email']
    self.pay_with = data['pay_type']
    place_order
  end
end
```

We have added two things here. The first is the `DEFAULT_DATA` hash that holds the default values to be used. The second is the first line of the `complete_order` method. This line now merges the hash being passed to the method with the `DEFAULT_DATA` hash. Where it finds an entry with a matching key it will use the value passed in.

With these changes our new *Scenario* works. But what if we do not want to provide any data? Let's write that *Scenario* now.

```
  Scenario: purchase one book with al default data
    When I press the Add to Cart button
    And I complete the order
    Then I should see "Thank you for your order"
```

Our new step definition looks like this.

```
And /^I complete the order$/ do
  @checkout = @shopping_cart.goto_checkout_page
  @checkout.complete_order
end
```

Notice that I am calling the `complete_order` method but this time I am passing now arguments. We have already seen how this works so the updated method should be no surprise.

```
def complete_order(data={})
  data = DEFAULT_DATA.merge(data)
  self.name = data['name']
  self.address = data['address']
  self.email = data['email']
  self.pay_with = data['pay_type']
  place_order
end
```

Our solution was to change the parameter to use a default value of an empty hash. When this method is called passing no parameter an empty hash is used. When merged with the default values we end up with the default values.

We now have the ultimate flexibility with our page object. We can provide as much or as little data as we see fit and other data will be added by the defaults.

Appendix B

RSpec Matchers