```
AgentGateway do |workshop|
 workshop :type => 'cucumber'
end
```

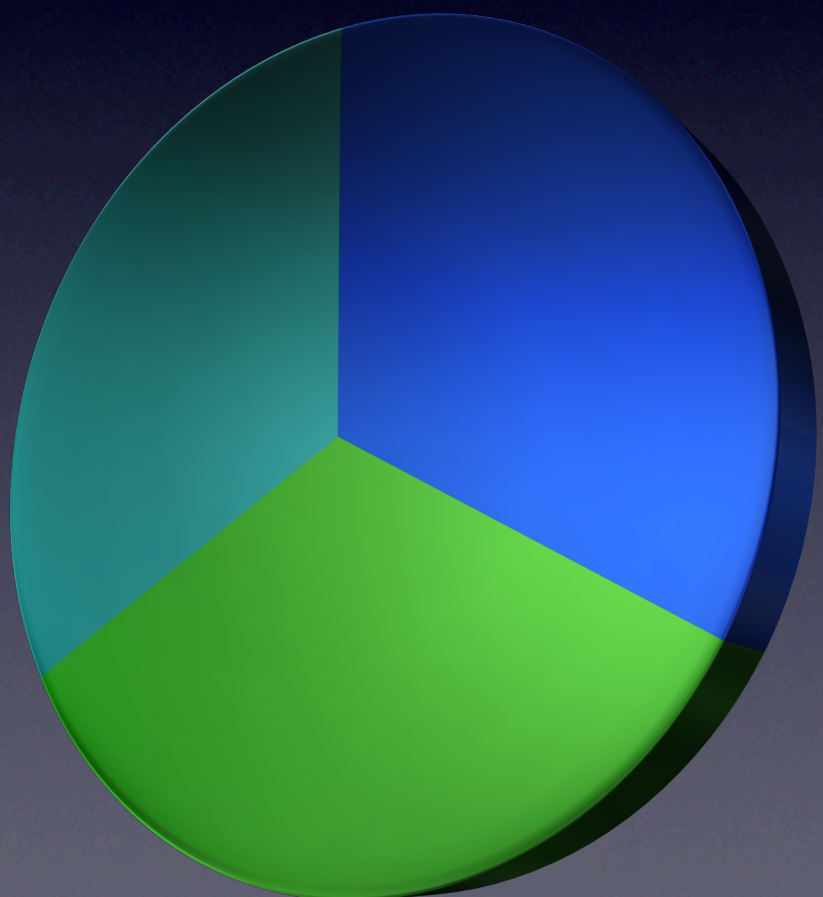
Current Process

Rework is Waste!

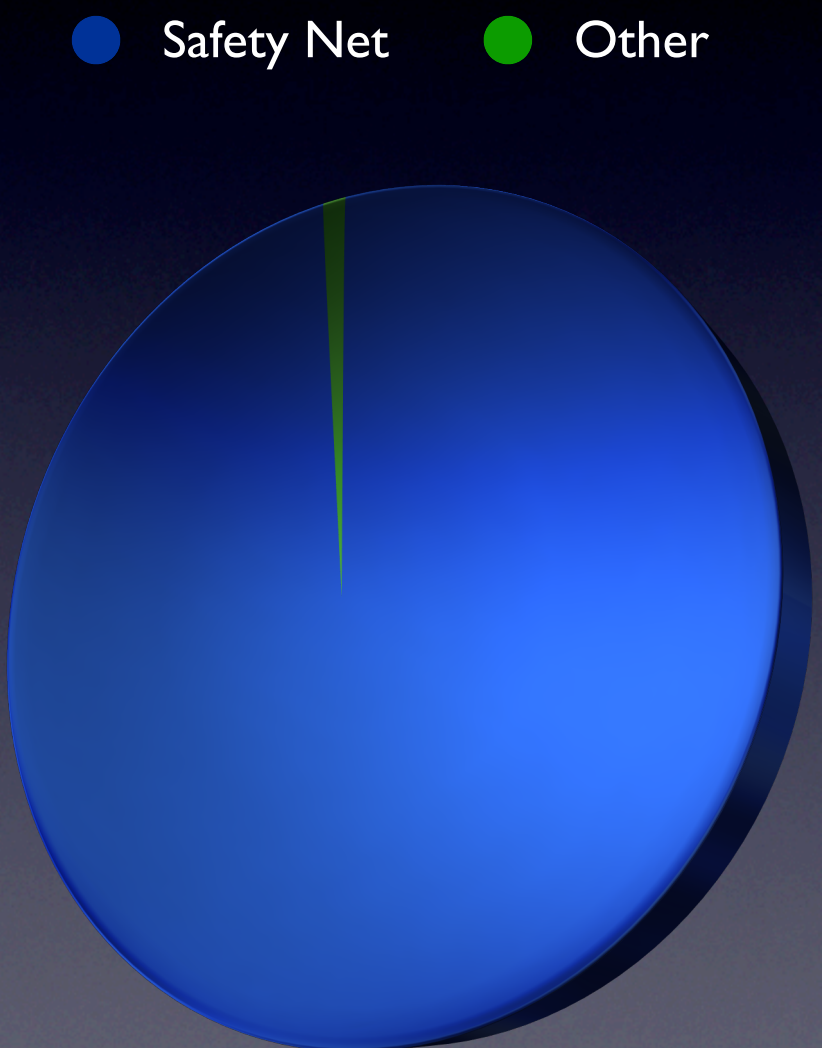
- It costs an industry average of $5,000 to fix one defect, due to the rework involved.

Why we say we test

- Report Progress
- Inform Stakeholder Decisions
- Learn



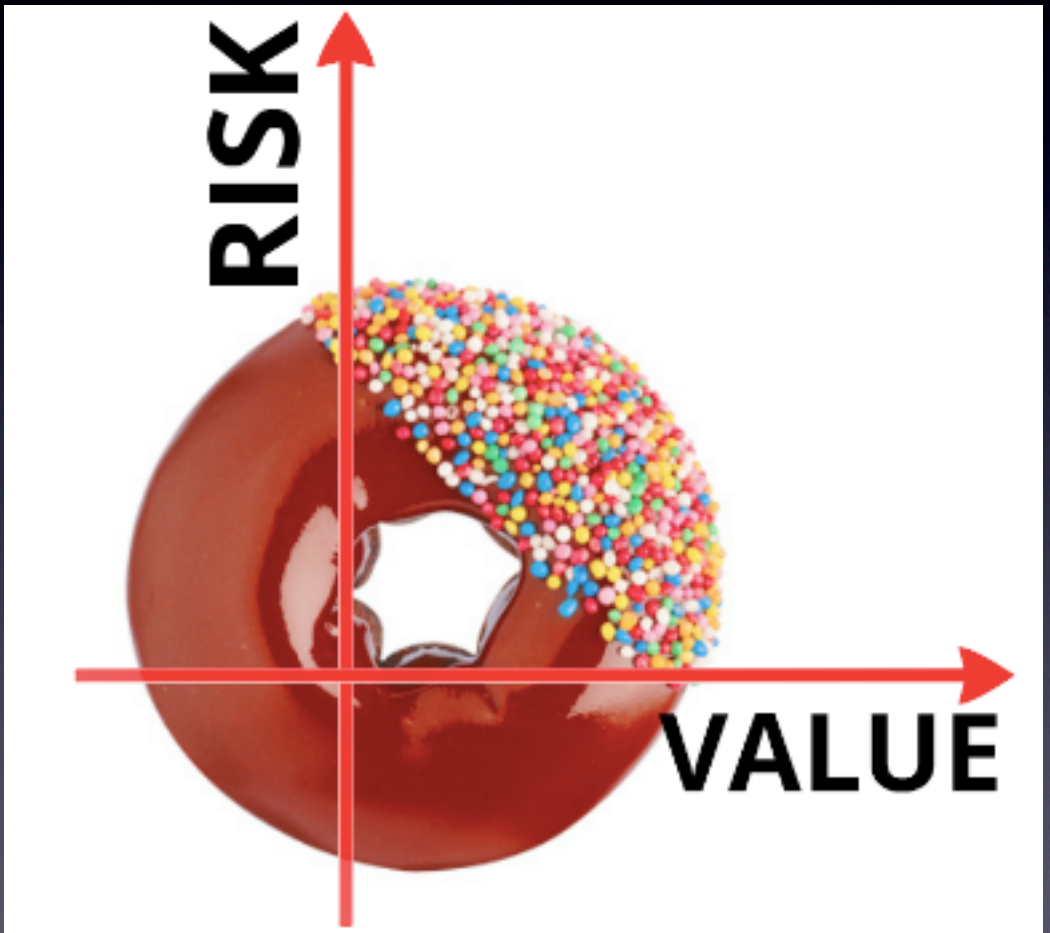
What actually happens



Traditional Testing is doomed to miss errors

- Manual script execution put the emphasis on low-value, low-risk work
- We find defects, but only after the cost of fixing them has gone up considerably
- We limit our available time to testing easy scenarios, and transient errors and corner cases go undetected for a long time.

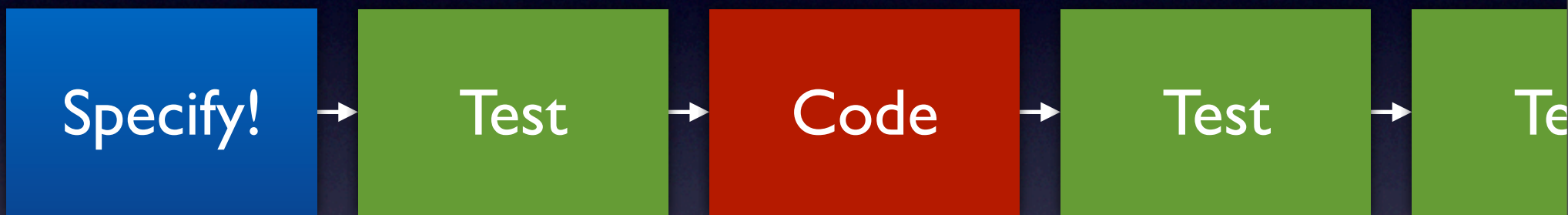
If you could only take one bite...



How do we get out of this world?

- We use software tools that let the machines do the low-value low-risk work, and don't require humans to execute them.
- We become highly engaged with the Product Owners and Developers to put the brain of the tester up front where it can help the most
- This is called "Specification by Example"

The New Process

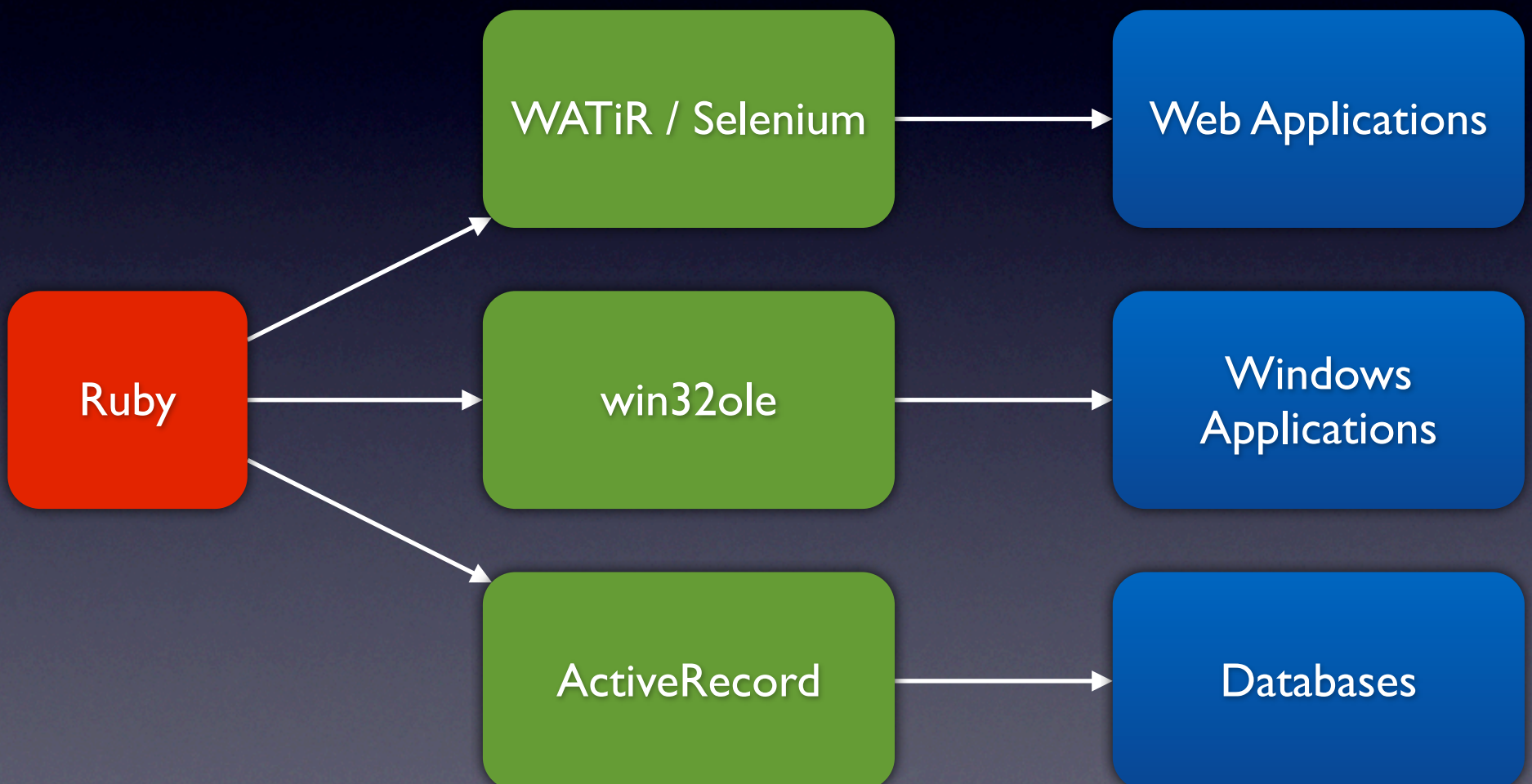


Introducing Ruby

- Ruby is a programming language publicly introduced in 1995, and has become massively popular in the last 5 years
- Free, Object-Oriented and Open Source, Ruby can be used for almost anything

Testing with Ruby

Ruby can be used to test almost anything



Introducing WATiR

- Stands for "Web Application Testing in Ruby"
- Pronounced "Water"
- Provides a simple domain-specific language to drive a browser to interact with web-pages

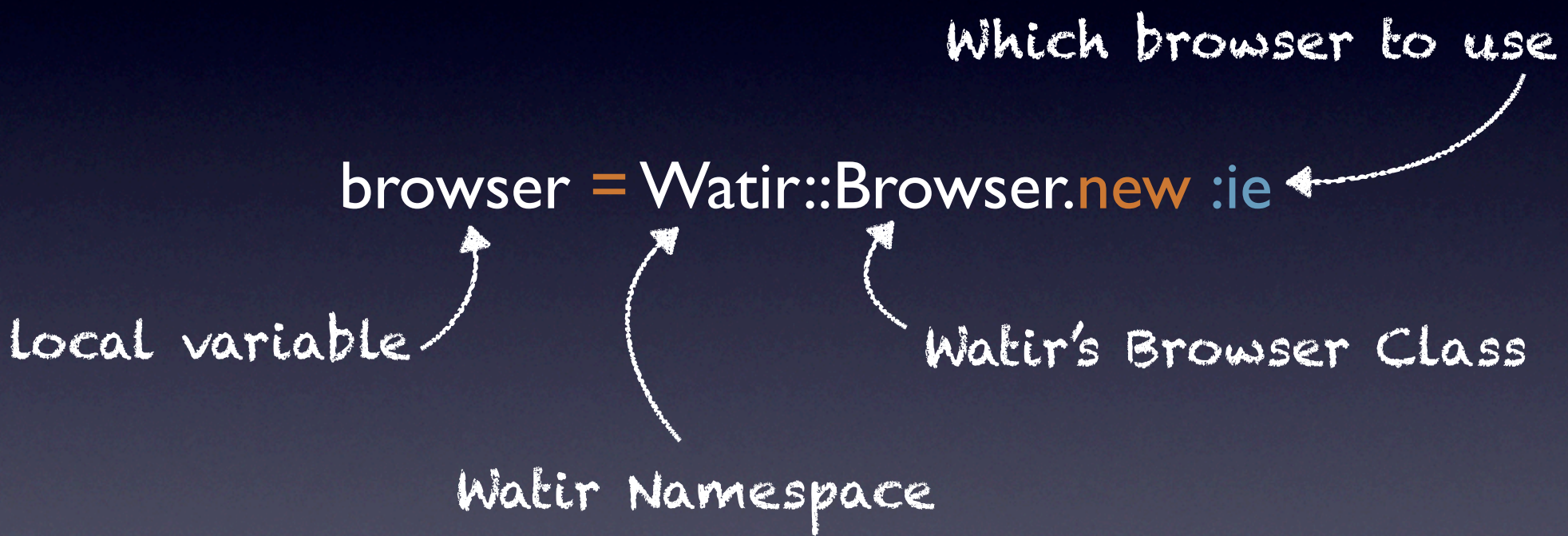
Using WATiR

Create a new script in the scripts directory of test_puppies called adopt_a_puppy.rb

```
require 'rubygems'
require 'watir-webdriver'
```

Interact with a browser

Create a browser object!



Tell the browser what to do

```
browser.goto “http://www.google.com”
browser.text_field(:name => ‘username’).set(‘Bradley’)
browser.button(:value => ‘Click here’).click
browser.checkbox(:name => ‘enabled’).set
browser.link(:href => ‘http://www.google.com’).click
browser.table(:name => ‘quotes’)[1][0]
```

Test!

Get text from the status bar

```
browser.status
```

Get the text from a page or element

```
browser.text
```

Get the html from a page or element

```
browser.html
```

Return true if the text appears on the page

```
browser.text.include? 'Agent Gateway'
```

Assert something

```
if browser.text.include? 'Mickey Mouse'
   puts 'Test Passed'
else
   puts 'Test Failed'
   fail
end
```

outputs string to the command line

Script reports itself as failed

Ruby emphasized highly readable code

```
fail unless browser.text.include? ‘Mickey Mouse’
```

Anyone can figure out what this code does.

Exercise!

- Write a WATiR script that adopts one puppy
- Use IE8 or Chrome Developer Tools to identify the names, ids and values with which to identify objects

Stay D.R.Y.!

- Don't Repeat Yourself!
- If you do something more than once, it should be encapsulated in a method so you can use it again later
- It's okay if a UI change breaks 40 scripts, but we should only have to change one thing to fix it

Defining a method in Ruby

```
def go_to_the_puppy_store
  @browser.goto "http://localhost:3000"
end
```

browser vs. @browser

- Defining a variable with nothing before it makes it a local, scoped variable. It's only available in the block of code in which you declared it.
- putting an @ before the variable name makes it an instance variable, which makes it available during your entire script

Exercise! Stay D.R.Y.!

- Go through your puppy adoption script and move your code to methods!

Being more specific

- Up until now, our puppy adoptions have just adopted the first puppy
- WATiR supports multiple selectors, so we can get more specific

Selecting by name and index

```
@browser.button(:value => 'View Details', :index => 0).click
```

Symbols

- Ruby uses a data type called a symbol frequently. They look like this: `:value`
- Symbols are more or less like strings, except they are **immutable** and are **Singletons** (They only ever exist in memory once)
- They get used frequently to assign attributes to ruby objects.

Hashrockets

- Hashrockets are the => syntax. They get used to assign values to hashes (key-value pairs)
- In the code: `:value => 'View Details'` the string 'View Details' is being assigned as the value to the value key, which is represented by a symbol!

Modules!

- So our adopt a puppy script has been methodized so that we can easily repeat steps, but right now, we can't use those steps in other scripts.
- We need a module!

Defining a module

- Create a new file in the scripts directory. Call it adoption_helper.rb

```
module AdoptionHelper
  def go_to_the_puppy_store
    ...
  end
end
```

Using a module (Ruby 1.8.7)

Ruby constant, referring to the current file

```
require File.dirname(__FILE__) + '/adoption_helper.rb'
include AdoptionHelper
```

appending the relative path to the module file

This isn't required, but if we don't include it, we have to type AdoptionHelper every time we want to call a method

Ruby 1.9.2+ only

A new method used for requiring a file relative to the current file

```
require_relative 'adoption_helper.rb'
include AdoptionHelper
```

Not backwards compatible with Ruby 1.8!

Exercise

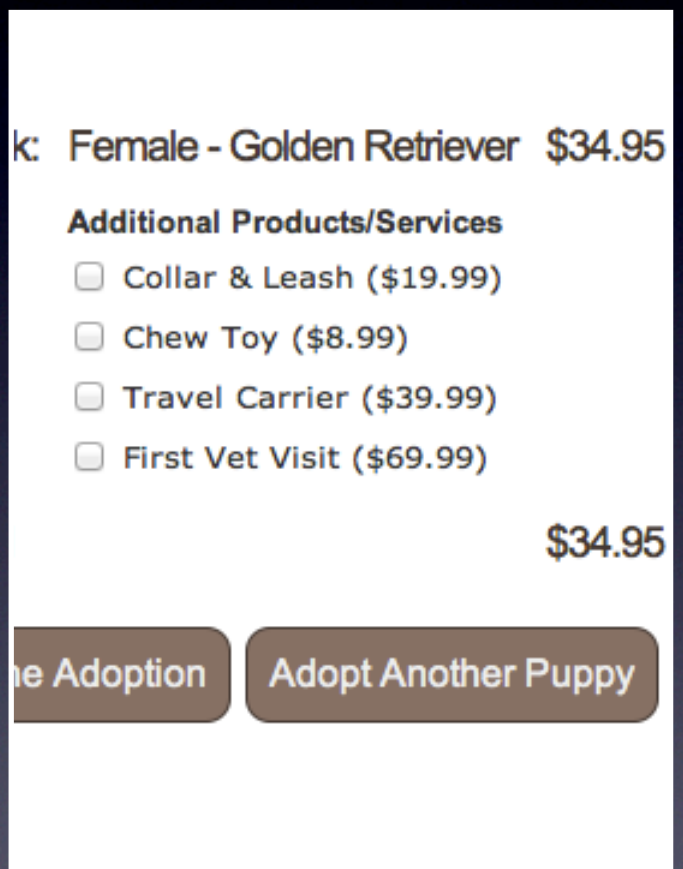
- Move all the adopt_a_puppy.rb methods into a module, and use that module in adopt_a_puppy.rb

Adopting Two Puppies

- We're still only adopting one puppy.
- How is the flow different for two puppies?

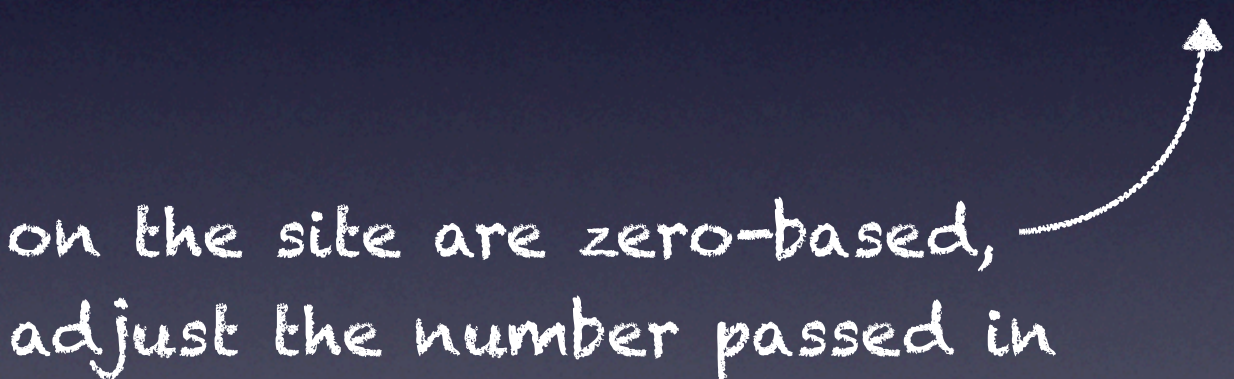
Two Puppies

- The view details button method will need to respond to a specific puppy
- We need to press Adopt Another Puppy instead of Complete the Adoption



Puppies by the Numbers!

```ruby
def adopt_puppy_number(num)
  @browser.button(:value => "View Details", :index => num-1).click
end
```



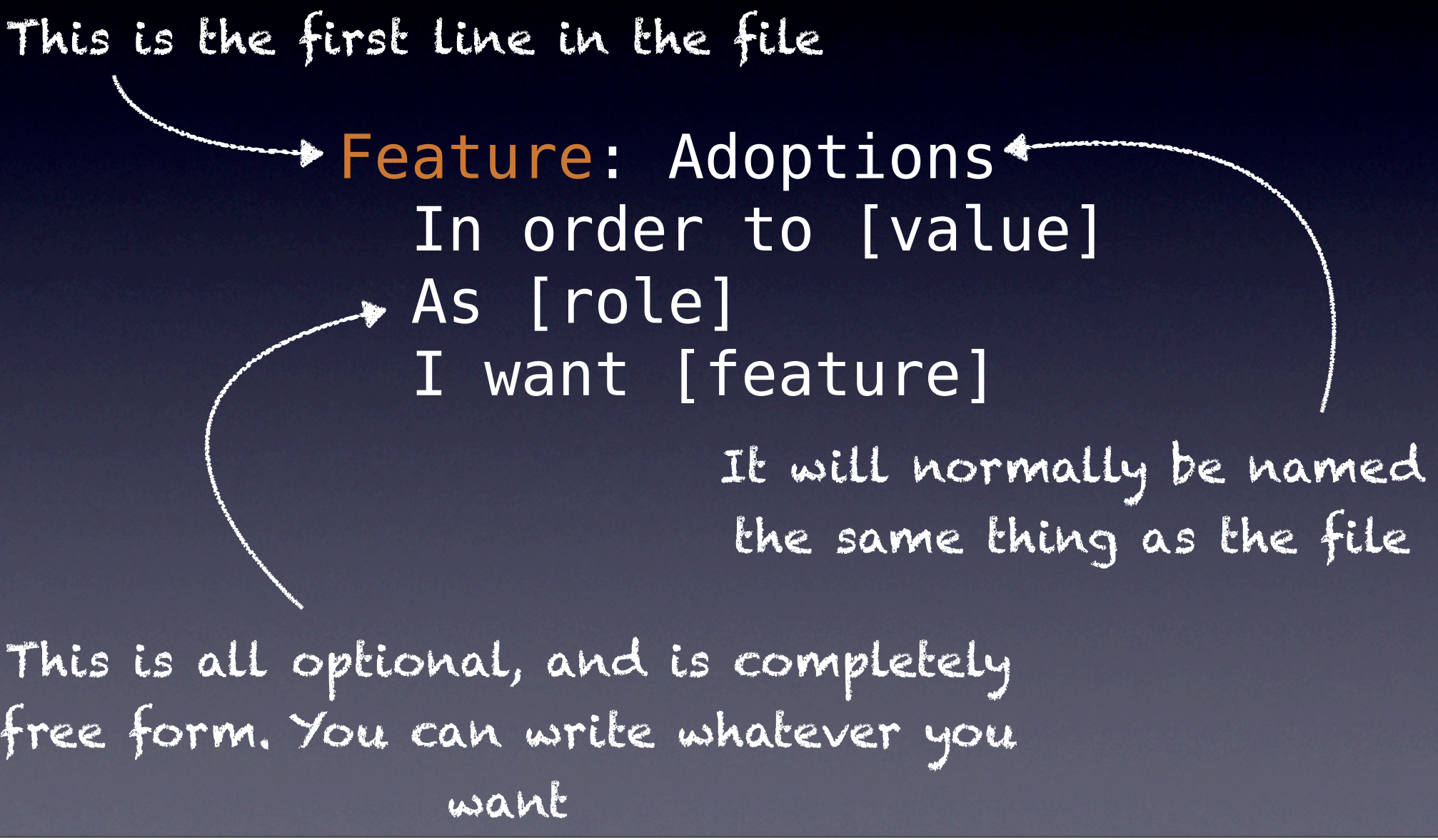
Exercise

- Create a new file in the scripts directory called adopt_two_puppies.rb
- Use the adoption_helper.rb module to adopt two puppies
- Don't forget to stay D.R.Y.

Introducing Cucumber

- The cucumber tool takes a feature file, written in plain text, and matches it to a block of code.
- The language feature files are written in is called Gherkin

Defining a Feature



Scenarios

- The Feature Definition is meant for the product owner to write what the acceptance criteria is.
- The testers then write different scenarios of the system as it will be used.
- Each one is one example of the system under use, and there can be as many of them as are necessary.

Scenario Definition (Puppy Adoption)

```
Scenario: Adopt One Puppy
    Given I am on the puppy adoption page
    When I click the View Details Button
    And I click the Adopt Me Button
    And I click the Complete the Adoption Button
    And I enter "Bradley" in the name field
    And I enter "123 Main St" in the address field
    And I enter "templeb3@nationwide.com" in the email field
    And I select "Credit card" from the pay type dropdown
    And I press the Place Order button
    Then I should see "Thank you for adopting a puppy"
```

Given, When, Then

- Given sets up the test. Given provides the context for the test.
- When is the action you are taking. It's what you're doing to poke the application
- Then is your expected outcome. This is what the results of the When statement are.
- Each one can be extended with And and But statements

Given, When, Then, don't matter

- Given, When and Then aren't actually keywords in Gherkin. They make a step easier to read, but the interpreter actually ignores them. You can replace them with * if you don't want them

```
Scenario: Adopt One Puppy
    * I am on the puppy adoption page
    * I click the View Details Button
    * I click the Adopt Me Button
    * I should see "Thank you for adopting a puppy"
```

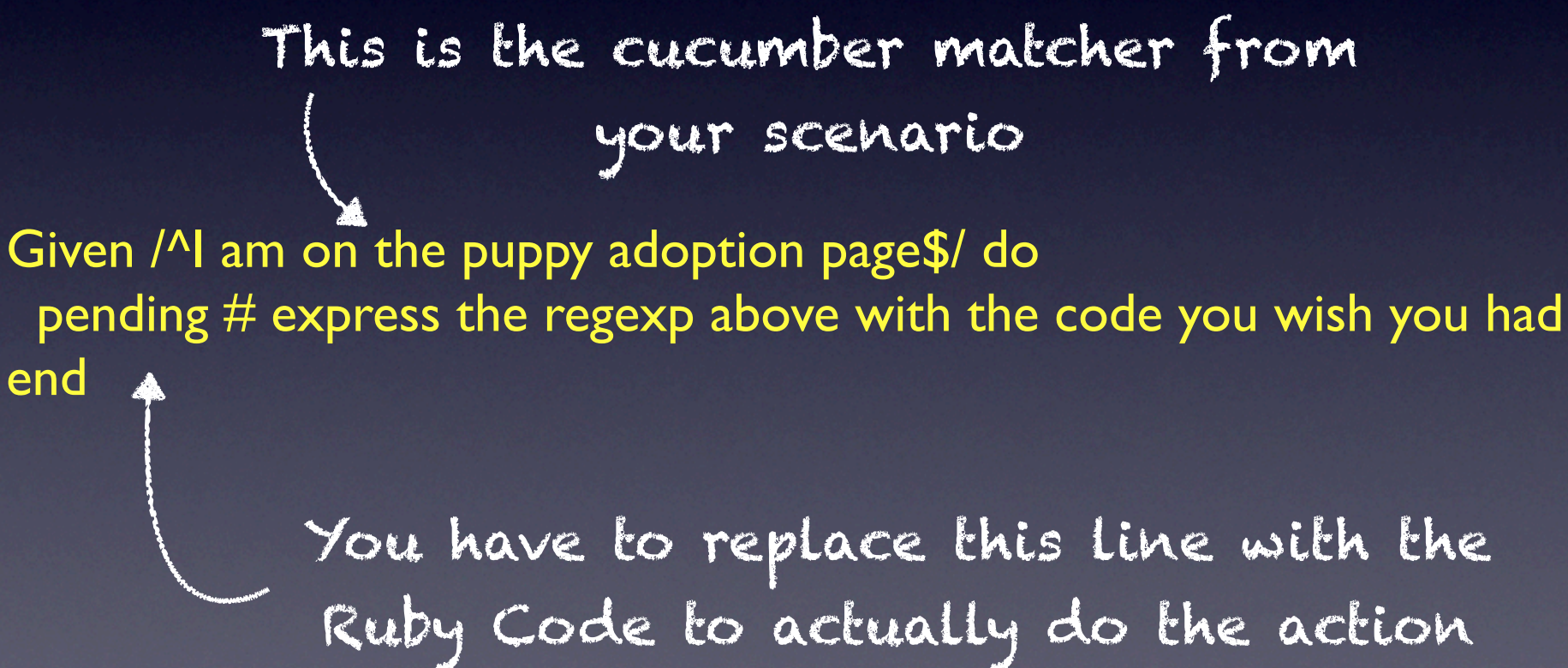
Run the Scenario

- Execute the specification as it is right now.

```
Scenario: Adopt One Puppy
    Given I am on the puppy adoption page
    When I click the View Details Button
    And I click the Adopt Me Button
    And I click the Complete the Adoption Button
    And I enter "Bradley" in the name field
    And I enter "123 Main St" in the address field
    And I enter "templeb3@nationwide.com" in the email field
    And I select "Credit card" from the pay type dropdown
    And I press the Place Order button
    Then I should see "Thank you for adopting a puppy"
```

Stubs

- When running the scenario, the command line will begin to report back stubs like this:

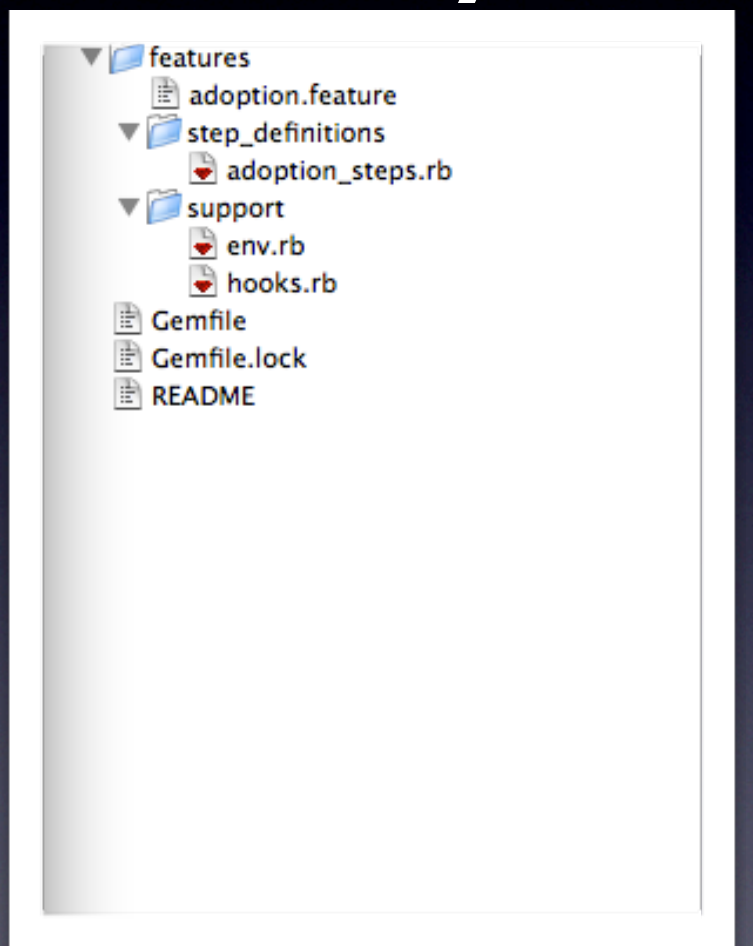


Step Definitions

- In the test puppies directory, under the features folder, create a new folder called step_definitions, if it isn't there already
- Create a new file in this folder called adoption_steps.rb

A quick tour around the features directory

- You have a feature, an empty step definition file, and a support directory with two ruby documents
- Env.rb is everything that we have pre-loaded into Cucumber. Hooks are some things that happen before and after every test.



hooks.rb

Before each test, we'll start up a browser with Watir

```
require 'watir-webdriver'
Before do
  @browser = Watir::Browser.new :firefox
end
After do
  @browser.close
end
```

And after each test we'll close that browser

env.rb

These are some of the libraries we're going to be using.

```
require 'rspec-expectations'
require 'page-object'
require 'page-object/page_factory'

World(PageObject::PageFactory)
```

World is the object that the step definitions run in. You can add things to World with syntax like this

Exercise

- Put the stub step definitions into adoption_steps.rb
- Use WATiR code to make the steps a reality
- Don't use the module, execute the code in the step definitions themselves

Steps with Parameters

This is called a "capture group"

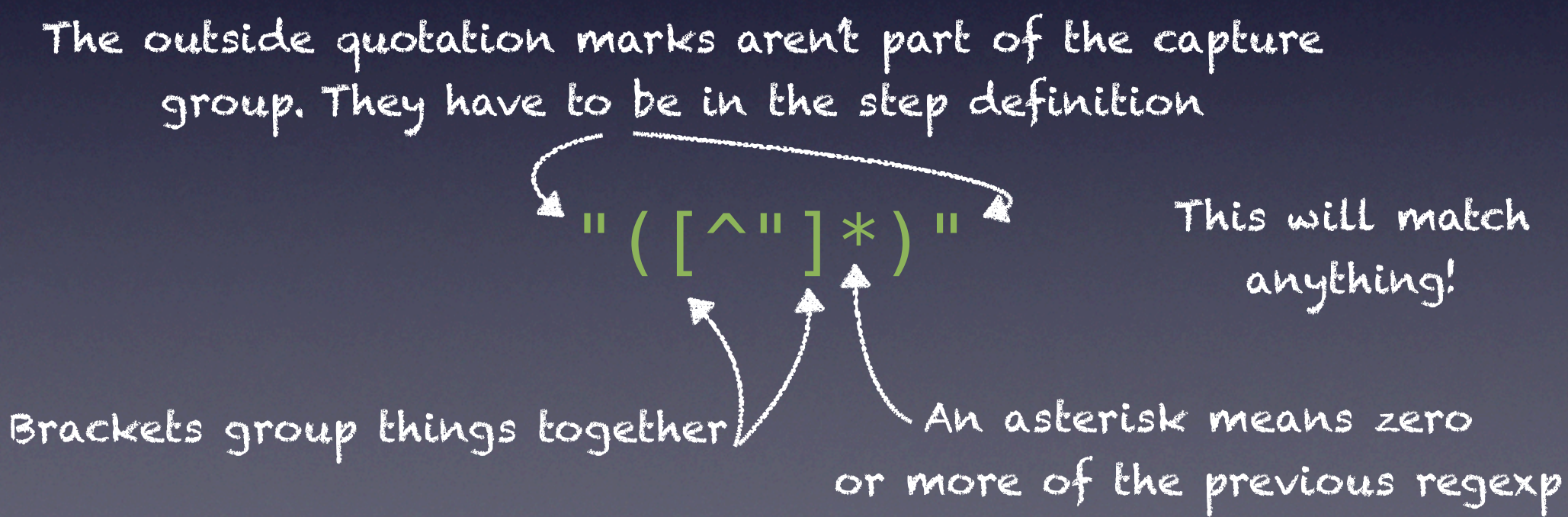
It takes whatever is inside of it and passes it to the code block

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

Which can then be used to do something in the browser

Capture Groups

- What counts is what's between the parentheses.
- Uses Ruby Regular Expression Syntax



Useful RegExp

RegExp	Matches
.	Any one character
...	Any 3 characters
.*	Zero or more characters
.+	One or more characters (anything except newline)
\$.*	Zero or more characters after a dollar sign
[0123456789]	Any Number
[0-9]	Any Number
[a-zA-Z]	Any Letter, lowercase or capital

Special RegExp

RegExp	Matches
\d	any digit
\w	Any word character
\s	Whitespace
\b	Word Boundary

Capitalizing these will match the opposite, so \D will mean any non-digit

Write your own capture groups

```
And I enter "123 Main St" in the address field
```

Matches 1+ numbers...

followed by 1+ other characters

```
/^And I enter "([\d+][.+])" in the address field$/
```

Keep it simple!

Background

- Sometimes a given feature will have a common set of tasks that need to be done first.
- For instance, most enterprise applications will require you to be logged in to test everything.
- Stay D.R.Y. Put those steps in a background!

adoptions.feature

Put this after the Feature declaration, but before the first scenario

```
Background:
  Given I am on the puppy adoption page
```

You can now remove this from your scenarios, because it will run for every scenario in the feature

Scenario Outline

- Sometimes you want to run through a scenario multiple times but change up some of the input
- You can define a Scenario Outline, and give the data you want to use in a table

Scenario Outline Adopt a Puppy

```
Scenario Outline: Adopt One Puppy
    When I click the View Details Button
    And I click the Adopt Me Button
    And I click the Complete the Adoption Button
    And I enter "<name>" in the name field
    And I enter "<address>" in the address field
    And I enter "<email>" in the email field
    And I select "<pay_type>" from the pay type dropdown
    And I press the Place Order button
    Then I should see "Thank you for adopting a puppy"

    Examples:
      | name             | address             | email              | pay_type       |
      | Bradley Temple   | 123 Main Street     | brad@example.com   | Check          |
      | Bob Dole         | 432 Roberts Road    | bob@example.com    | Credit card    |
      | George Costanza  | 18 Greenview Terrance | george@email.com  | Purchase order |
```

Instead of inputing the value you input the key in angle brackets

The first row of the table needs to correspond to the keys you provided in the scenario outline

Exercise

- Convert the Puppy Adoption scenario into a scenario outline
- You might find some clues on the previous slide!

Cucumber Arguments

- Cucumber arguments always come as strings, so if you need them to be some other data type, they need to be converted!

```
When /^I click the view details button for puppy number (\d+)$/ do |num|
  @browser.button(:value => 'View Details', :index => num.to_i-1).click
end
```

to_i converts the string to an integer, so we can use it like we'd expect

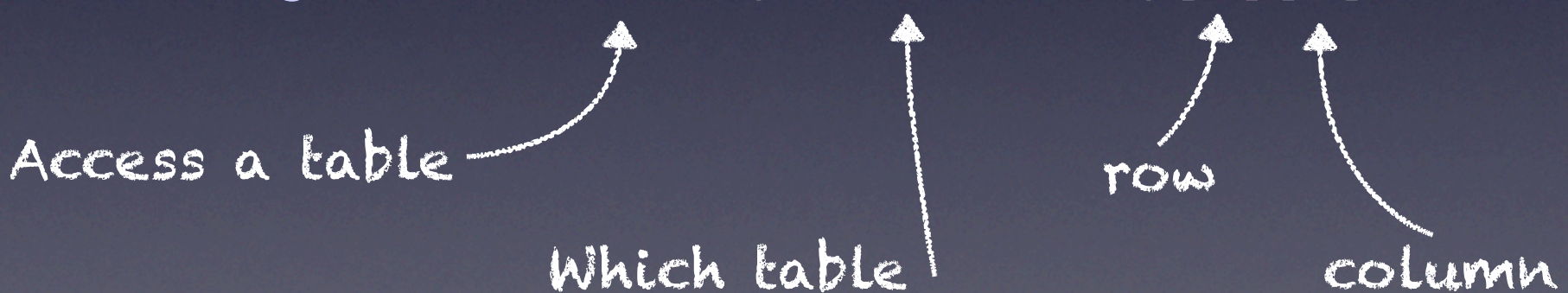
Exercise 8

- Make a new scenario for adopting two puppies

HTML Tables

- Tables are one of the more difficult elements to work with.
- To reference a particular cell, you have to access it via a 2-dimensional array

```
@browser.table(:index => 0)[0][1]
```



Exercise

- Create a scenario to validate the shopping cart page
- It should work for adopting a single puppy
- It should validate the puppy's name, how much it costs and the total cost.

Exercise Scenario

```
Scenario: Adopt One Puppy with validations
      When I click the view details button for puppy number 1
      And I click the Adopt Me button
      Then I should see "Brook" as the name
      And I should see $34.95 as the price
      And I should see $34.95 as the total price
```

We'll have to define these new steps

Exercise Step Definitions

Column 2 (index 1) has the name

Column 4 (index 3) has the price

```
Then /^I should see "([^"]*)" as the name$/ do |name|
  @browser.table(:index => 0)[0][1].text.should include name
end

Then /^I should see (\$\d*.\d\d) as the price$/ do |price|
  @browser.table(:index => 0)[0][3].text.should include price
end

Then /^I should see (\$\d*.\d\d) as the total price$/ do |price|
  @browser.td(:class => 'total_cell').text.should include price
end
```

The cell for the total price has a css class, so we can access it directly, no arrays needed

This only works for one puppy!

- How might it differ for two?

Adopt Two Puppies

```
Scenario: Adopt Two Puppies with Validations
      When I click the view details button for puppy number 1
      And I click the Adopt Me button
      And I click the Adopt Another Puppy button
      And I click the view details button for puppy number 2
      And I click the Adopt Me button
      Then I should see "Brook" as the name for puppy number 1
      And I should see $34.95 as the price for puppy number 1
      And I should see "Hanna" as the name for puppy number 2
      And I should see $22.99 as the price for puppy number 2
      And I should see $57.94 as the total price
```

Exercise

- Adopt Two puppies, verifying the names and prices for each, as well as the total price.

Solution

- There are six rows for each puppy
- By defining a method in our steps file, we can return the correct row for each puppy

```
def row_for_puppy_number(num)
  (num-1)*6
end
```

Don't forget it's zero based!

This will give us row 0 for puppy 1 and row 6 for puppy 2, 12 for 3, etc.

Step Definitions

```
Then /^I should see "([^"]*)" as the name for puppy number (\d+)$/ do |name, puppy_number|
  @browser.table(:index => 0)[row_for_puppy_number(puppy_number.to_i)][1].text.should include name
end

Then /^I should see (\$\d*.\d\d) as the price for puppy number (\d+)$/ do |price, puppy_number|
  @browser.table(:index => 0)[row_for_puppy_number(puppy_number.to_i)][3].text.should include price
end
```

Magic Numbers

- In programming, magic numbers are values that cause software to work... right now
- A change in design could cause magic numbers to change, and code to break
- Don’t use magic numbers!

Magic Puppy Numbers

- Our code uses three magic numbers right now
  - 6 Rows Per Puppy
  - Column 1 holds the name
  - Column 3 holds the subtotal
- How can we refactor?

Constants!

- By defining our constants at the top of the file, we can stay D.R.Y, as this gives us a single place to update for any changes

```
#constants
ROWS_PER_PUPPY = 6
SUBTOTAL_COLUMN = 3
NAME_COLUMN = 1
```

Exercise

- Refactor your code to use constants instead of magic numbers

Passing Data as a Table

- When you have a lot of data that may change, you don't need to define a complicated step.
- You can just pass the data in a table

Table Scenario Definition

```
Scenario: Using a table for data
  When I click the  view details button for puppy number 1
  And I click the Adopt Me Button
  And I click the Complete the Adoption Button
  And I checkout with:
    | name           | address     | email                      | pay_type |
    | Bradley Temple | 123 Main St | templeb3@nationwide.com | Check    |
  Then I should see "Thank you for adopting a puppy"
```

This will get passed to the step definition as an array of hashes

Using table in the step definition

We can only use one set of data, so we'll grab the first hash from the table

```
When /^I checkout with:$/ do |table|
  data = table.hashes.first
  @browser.text_field(:id => 'order_name').set(data['name'])
  @browser.text_field(:id => 'order_address').set(data['address'])
  @browser.text_field(:id => 'order_email').set(data['email'])
  @browser.select_list(:id => 'order_pay_type').select(data['pay_type'])
  @browser.button(:id => 'order_submit').click
end
```

And then we'll use that hash to fill out the form

Exercise

- Add a scenario to checkout with a passed in table!

Default Data

- In most web apps, the data you enter can end up being largely inconsequential
- You can save yourself some trouble by defining default data to use and then just specify the data of importance

Using a hash to specify default data

Goes at the top of Step Definitions with the constants

```
DEFAULT_DATA = {
  'name' => 'Bradley',
  'address' => '1548 Greenview Dr',
  'email' => 'templeb3@nationwide.com',
  'pay_type' => 'Check'
}
```

Keys

Values

Exercise

- Write a scenario step definition that uses all the default data

Staying D.R.Y.

- Looking over our step definitions, we're not doing a great job of staying D.R.Y.
- There are several places where we're doing the same things, if in a slightly different way. Is there a better solution?

Introducing PageObject

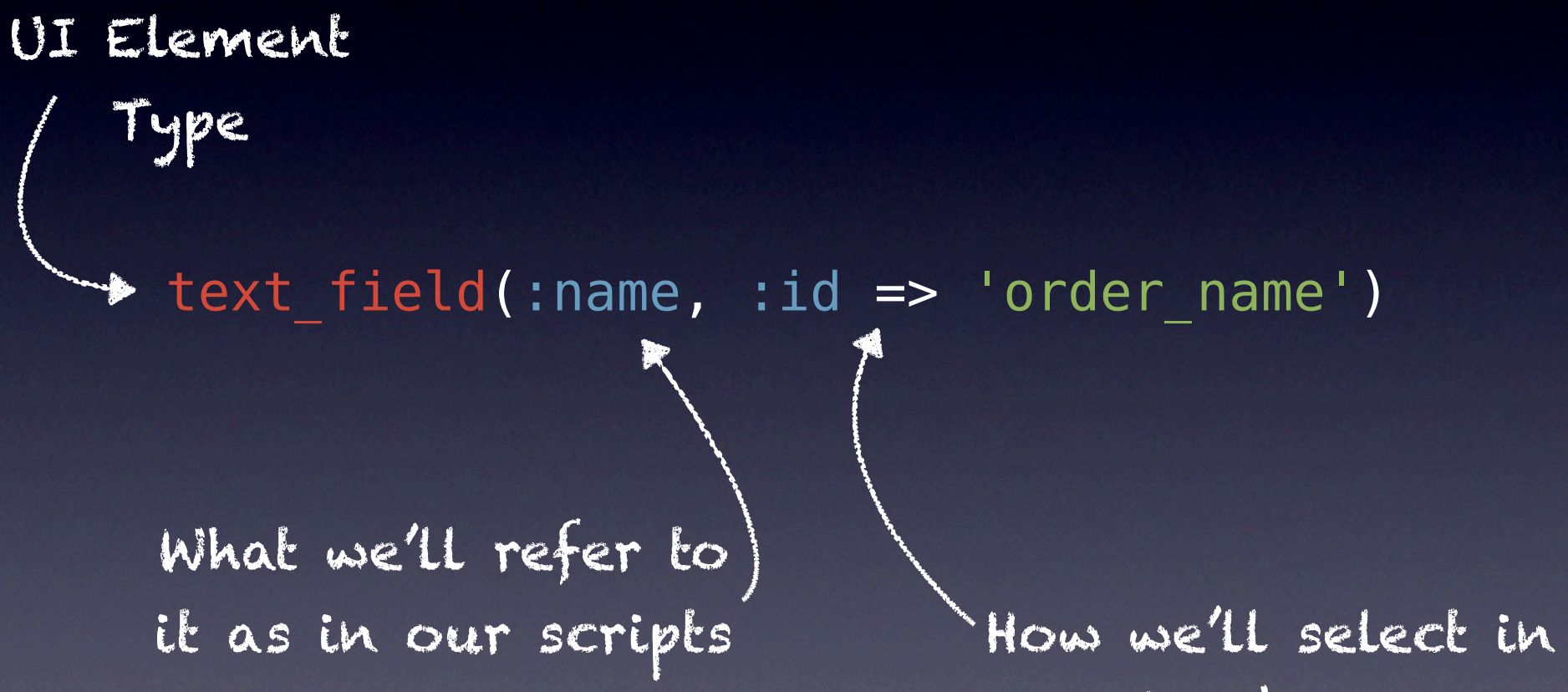
- PageObject is a design pattern that involves creating a class that represents a page on our site
- It offers many time and code saving advantages, but more importantly, gives us a single place to go to update for any changes to our site
- Jeff "Cheezy" Morgan released a page-object gem for us to use.

Defining a PageObject

- Create a new folder under the support folder called pages
- Create a new file in the pages directory called home_page.rb

```
class HomePage
  include PageObject
  ...
end
```

Defining UI Elements



Making the Home Page a PageObject

- What elements do we interact with?

home_page.rb

page_url is only required if you want to navigate directly to the page

```
class HomePage
  include PageObject
  page_url "http://localhost:3000"
  button(:view_details, :value => 'View Details')
end
```

This works, but is only useful for adopting Brook! You can of course define a button for each dog, but maybe there is a better way

PageObject Methods

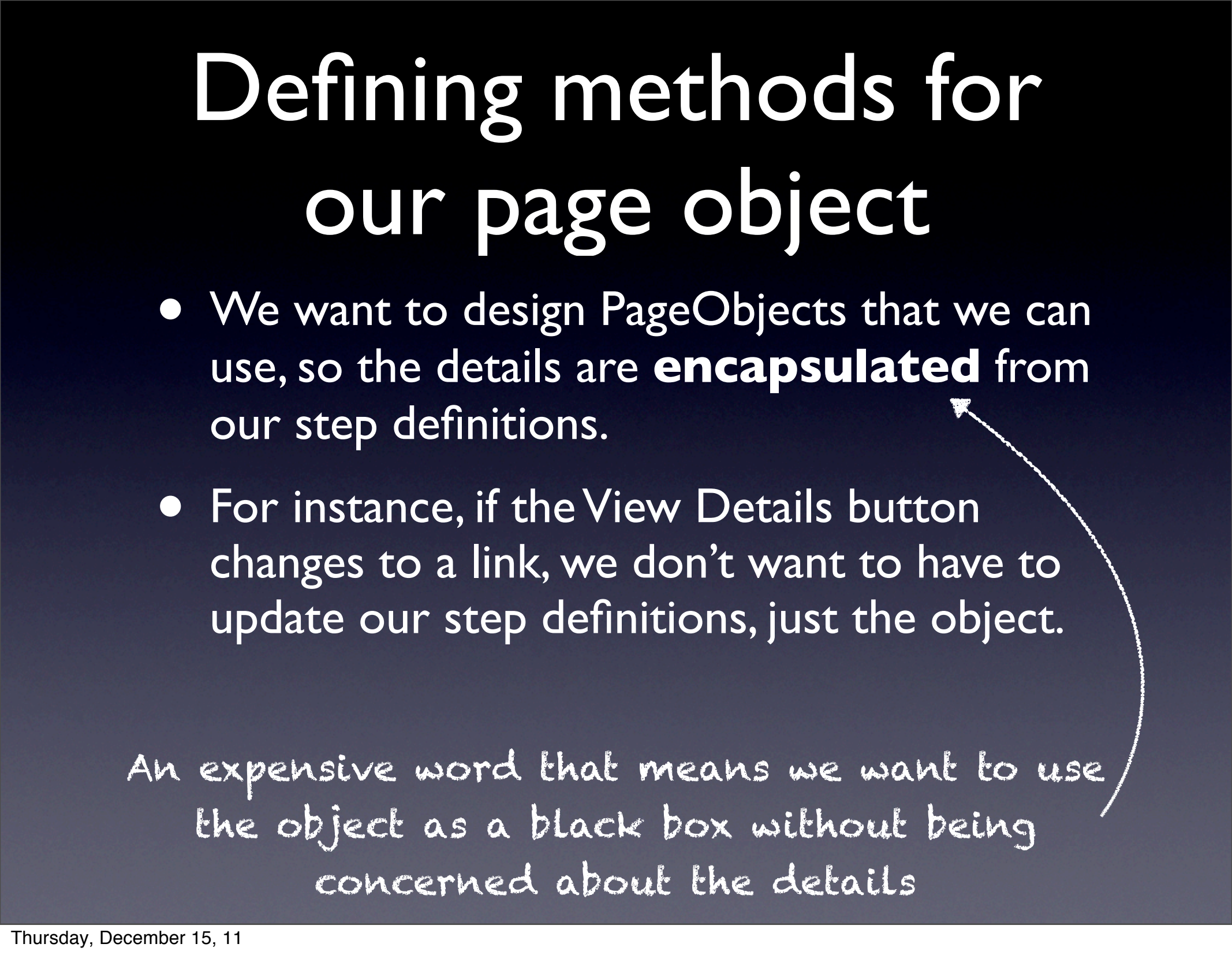
- Each page object generates three methods for each UI Element you define. For instance, if we defined a text field called username...
  - def username
  - def username=
  - def username_element

What do you mean “generates methods”?

- Ruby is an “interpreted language”, this means that things can happen at runtime that you didn’t code at all
- Libraries can implement certain actions to take that will actually write and execute code that the coder never defined, this allows for very flexible code

Defining methods for our page object

- We want to design PageObjects that we can use, so the details are **encapsulated** from our step definitions.
- For instance, if the View Details button changes to a link, we don't want to have to update our step definitions, just the object.



An expensive word that means we want to use the object as a black box without being concerned about the details

Adopting a Puppy

```
def adopt_puppy
  view_details_element.click
end
```

Gets the actual button element

And tells WATiR to click it!

Using the PageObject

- Two basic methods:
  - visit_page
  - on_page

Visit Page

- You only use visit_page if you want to directly access a URL, as defined in the object as page_url
- Any other time, you'll use a UI Element to access get to the page in question

```
visit_page(HomePage)
```

on_page

- Tell the page object what to do. You can use get/set text for a UI element, access it directly, or use a method you defined

on_page(HomePage).adopt_puppy

on_page multiple actions

- If you are going to do multiple actions on one page, you can instead use a block

```
on_page(HomePage) do |page|
  page.do_this
  page.do_that
end
```

We're still only adopting the first puppy

- How might we adopt a puppy by name?
- How about a hash mapping puppy names to their indexes?

```
NAME_LOOKUP = {
  'Brook' => 0,
  'Hanna' => 1
}
```

Remember, these are the keys

and these are values

Defining a name lookup method

- We can make a private method for our page to lookup a puppy by name

```
private
def index_for(name)
  NAME_LOOKUP[name]
end
```

This keyword means what's below can only be accessed from within this class

Exercise

- Using the code on the previous slides, let's refactor our HomePage PageObject to be able to adopt a puppy by name.

Solution

```
class HomePage
  include PageObject

  NAME_LOOKUP = {
    'Brook' => 0,
    'Hanna' => 1
  }

  page_url "http://localhost:3000"

  def adopt_puppy(name)
    button_element(:value => 'View Details', :index =>
index_for(name)).click
  end

  private
  def index_for(name)
    NAME_LOOKUP[name]
  end
end
```

This code is on the tricky side. Instead of defining the button up front, we're accessing it dynamically to avoid repetition

Exercise

- Refactor your feature and step definitions to lookup the puppies by name and use the HomePage PageObject

Refactoring Step 1: Feature

- Changing the lookup by numbers to names and running Cucumber will give us a new stub:

```
When /^I click the view details button for "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

- Lets Implement it!

Refactoring Step 2: Step Definitions

- We have to update our Background step to use the PageObject, then implement the new one

```
Given /^I am on the puppy adoption page$/ do
  visit_page(HomePage)
end

When /^I click the view details button for "([^"]*)"$/ do |name|
  on_page(HomePage).adopt_puppy name
end
```

Refactoring Step 3: Out with the Old

- We don't adopt any puppies by number anymore, so don't keep old code that isn't useful hanging around in your step definition file. Delete it!

```
When /^I click the view details button for puppy number (\d+)$/ do |num|
  @browser.button(:value => 'View Details', :index => num.to_i-1).click
end
```

Exercise

- Let's start defining more PageObjects
- The Puppy Details Page is our next stop
- Hint: You should only need one line of code, aside from the class declaration and include.

Solution

```
#features/support/pages/puppy_detail_page.rb
class PuppyDetailPage
  include PageObject
  button(:adopt_me, :value => "Adopt Me!")
end

#features/step_definitions/adoption_steps.rb
When /^I click the Adopt Me button$/ do
  on_page(PuppyDetailPage).adopt_me
end
```

Exercise

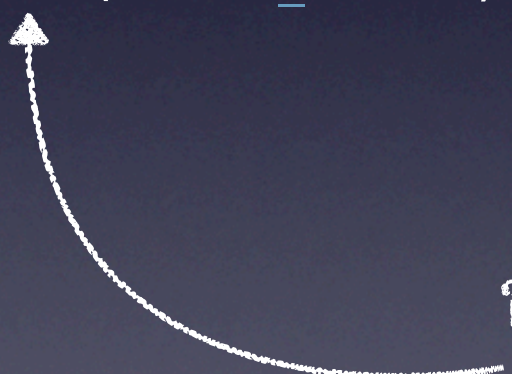
- Develop a Page Object for the shopping cart page.
- Move your existing logic from the step definition to the new page object, the refactor the step definitions to use it

Solution

```
class ShoppingCartPage
  include PageObject
  #constants
  ROWS_PER_PUPPY = 6
  NAME_COLUMN = 1
  PRICE_COLUMN = 3
  ...
end
```

Define elements

```
button(:complete_adoption, :value => 'Complete the Adoption')
button(:adopt_another, :value => 'Adopt Another Puppy')
table(:cart, :index => 0)
cell(:cart_total, :class => 'total_cell')
```



PageObject uses the more descriptive word 'cell' instead of 'td'

Refactor Steps

```
Then /^I should see (\$\d*.\d\d) as the total price$/ do |price|
  on_page(ShoppingCartPage).cart_total.should == price
end

When /^I click the Complete the Adoption button$/ do
  on_page(ShoppingCartPage).complete_adoption
end

When /^I click the Adopt Another Puppy button$/ do
  on_page(ShoppingCartPage).adopt_another
end
```

Now for the tricky steps...

- We still have the name for puppy number and price for puppy number steps.
- We probably just want to give the page the puppy number and ask it for the name and price

Designing the methods

- We ultimately need to end up with this:

```
cart_element[0][1]
```

- Let's create a helper method to give us the row for a particular number

row_for

This is just to help us use the page, so mark it private

```
private
def row_for(line_item)
  (line_item - 1) * ROWS_PER_PUPPY
end
```

Just like before, we need to adjust the number to be 0-based

Now the public methods

```
def name_for(line_item)
  cart_element[row_for(line_item)][NAME_COLUMN].text
end

def subtotal_for(line_item)
  cart_element[row_for(line_item)][PRICE_COLUMN].text
end
```

Gets the actual table object

Gets the row for the correct puppy

Gets the appropriate column

Now update our last step definitions

```
Then /^I should see "([^"]*)" as the name for puppy number (\d+)$/ do |name,
puppy_number|
  on_page(ShoppingCartPage).name_for(puppy_number.to_i).should include name
end

Then /^I should see (\$\d*.\d\d) as the price for puppy number (\d+)$/ do |price,
puppy_number|
  on_page(ShoppingCartPage).subtotal_for(puppy_number.to_i).should == price
end
```

Exercise

- There's only one page left to convert to a PageObject
- Create a checkout_page.rb!

CheckoutPage

```
class CheckoutPage
  include PageObject

  DEFAULT_DATA = {
    'name' => 'Bradley',
    'address' => '123 Main St',
    'email' => 'templeb3@nationwide.com',
    'pay_type' => 'Check'
  }
  ...
end
```

Bring over the default data from the step definitions

UI Elements

```
text_field(:name, :id => 'order_name')
text_field(:address, :id => 'order_address')
text_field(:email, :id => 'order_email')
select_list(:pay_type, :id => 'order_pay_type')
button(:place_order, :id => 'order_submit')
```

Refactoring Steps

- How can we change this to be D.R.Y. and easy to read?

```
When /^I checkout with:$/ do |table|
  data = table.hashes.first
  @browser.text_field(:id => 'order_name').set(data['name'])
  @browser.text_field(:id => 'order_address').set(data['address'])
  @browser.text_field(:id => 'order_email').set(data['email'])
  @browser.select_list(:id => 'order_pay_type').select(data['pay_type'])
  @browser.button(:id => 'order_submit').click
end
```

A method with a default value

In Ruby, you can give a method a default value. In this case an empty hash.

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

self makes these calls on the current object, the CheckoutPage

We'll merge whatever gets passed in with our default data

New Step Definitions

```
When /^I checkout with:$/ do |table|
  on_page(CheckoutPage).complete_order(table.hashes.first)
end

When /^I checkout with default data$/ do
  on_page(CheckoutPage).complete_order
end
```

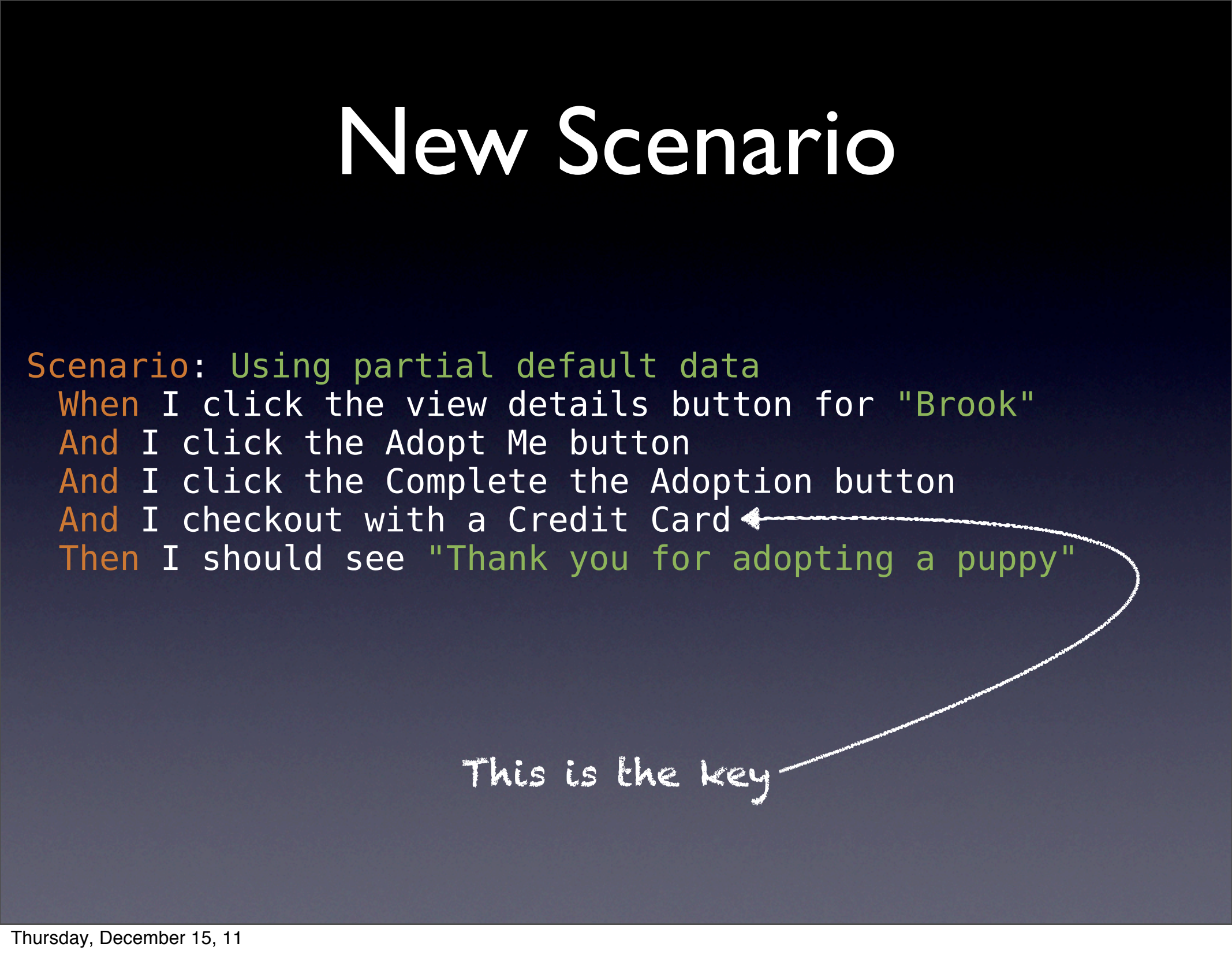
Since we don't pass anything, the hash will just get defaulted to empty and pick up the default values we defined

Using partial default data

- With our default data scheme all set up, we can now write a feature using partial data.

New Scenario

```
Scenario: Using partial default data
  When I click the view details button for "Brook"
  And I click the Adopt Me button
  And I click the Complete the Adoption button
  And I checkout with a Credit Card
  Then I should see "Thank you for adopting a puppy"
```



New Step Definition

```
When /^I checkout with a Credit Card$/ do
  pending
end
```

Exercise

- Implement the new step definition without writing any new code in CheckoutPage

Solution

```
When /^I checkout with a Credit Card$/ do
  on_page(CheckoutPage).complete_order({'pay_type' => 'Credit card'})
end
```

Since this method merges what's passed in with what we default, we can just pass in the hash of what we want and not worry about everything else

Exercise

- There's only a handful of step definitions still not using PageObject. Refactor them!

Solution

```
When /^I enter "([^"]*)" in the name field$/ do |name|
  on_page(CheckoutPage).name = name
end
When /^I enter "([^"]*)" in the address field$/ do |address|
  on_page(CheckoutPage).address = address
end
When /^I enter "([^"]*)" in the email field$/ do |email|
  on_page(CheckoutPage).email = email
end
When /^I select "([^"]*)" from the pay type dropdown$/ do |pay_type|
  on_page(CheckoutPage).pay_type_element.select pay_type
end
When /^I press the Place Order button$/ do
  on_page(CheckoutPage).place_order
end
Then /^I should see "([^"]*)"$/ do |text|
  @current_page.text.should include text
end
```

@current_page asks PageObject for whatever page you are on, which makes this step D.R.Y. and reusable

Why look at that

- Our Step Definitions are now very clean and concise.

```
Given /^I am on the puppy adoption page$/ do
  visit_page(HomePage)
end

When /^I click the view details button for "([^"]*)"$/ do |name|
  on_page(HomePage).adopt_puppy name
end

When /^I click the Adopt Me button$/ do
  on_page(PuppyDetailPage).adopt_me
end

Then /^I should see (\$\d*.\d\d) as the total price$/ do |price|
  on_page(ShoppingCartPage).cart_total.should == price
end

Then /^I should see "([^"]*)" as the name for puppy number (\d+)$/ do |name, puppy_number|
  on_page(ShoppingCartPage).name_for(puppy_number.to_i).should include name
end

Then /^I should see (\$\d*.\d\d) as the price for puppy number (\d+)$/ do |price, puppy_number|
  on_page(ShoppingCartPage).subtotal_for(puppy_number.to_i).should == price
end

When /^I click the Complete the Adoption button$/ do
  on_page(ShoppingCartPage).complete_adoption
end

When /^I click the Adopt Another Puppy button$/ do
  on_page(ShoppingCartPage).adopt_another
end
```

Now, finally, the right way

- Up to this point, we've done it completely wrong
- The features we've written, while instructive, are verbose and very brittle. A small change in the UI will break several features and require lots of rework
- Remember, Rework is Waste. Don't Repeat Yourself!

Adopt One Puppy

- Our very first feature describes every single click in input. The only thing it validates is that you see a thank you message.
- This Scenario Should be Two Statements

```
Scenario: Adopt One Puppy, The Right Way
  When I adopt a puppy
  Then I should see a thank you message
```

What's so great about that?

- It keeps things flexible. The UI can change day-to-day during development. The business focus of the project will not
- Your features should be written in a business-friendly language in the language of the business domain. Not calling out every click and data entry (unless there is business significance to them)

Making the Product Owner Care

- This process will only work if the product owner cares about the feature document. This is the living, breathing and executable version of their wants.
- Feature files filled with lots of technical details and jargon will cause them to ignore it. Keeping things in a common language will let them be involved

Exercise

- Implement the new steps for the feature written the right way.

```
When /^I adopt a puppy$/ do
  pending
end

Then /^I should see a thank you message$/ do
  pending
end
```

Solution

Now if the business flow changes, we only have to change it here.

You should also update adopt_puppy in HomePage to use a default name of Brook, so you don't have to pass a name in

```
When /^I adopt a puppy$/ do
  on_page(HomePage).adopt_puppy
  on_page(PuppyDetailPage).adopt_me
  on_page(ShoppingCartPage).complete_adoption
  on_page(CheckoutPage).complete_order
end

Then /^I should see a thank you message$/ do
  on_page(HomePage).text.should include "Thank you for adopting a puppy!"
end
```

```
def adopt_puppy(name='Brook')
```

Reusable Elements

- Web Pages aren't all unique. Most pages have at least some elements that are repeated elsewhere in the site.
- Menus, Headers, Footers, Error Message Sections, etc.
- We can define these as PageObjects and reuse them wherever we want

Error Messages

- The Checkout Page has validations that require the fields be filled out.
- We can just ask the whole page for it's text, that doesn't satisfy the scenario if the business rule requires it follows universal error message presentations, because text will return true no matter where in the page it is.

Let's write a scenario

```
Scenario: Name can not be blank message when checking out
     When I check out without entering name
     Then I should see the error message "Name can't be blank"
```

Exercise

- Implement the new scenario