
RUBY
BITS
BLOCKS, PROCS, AND LAMBDAS
PRESS START

REMEMBER THIS?

```ruby
def call_this_block_twice
  yield
  yield
end
```

```ruby
call_this_block_twice { puts "tweet" }
```

```
tweet
tweet
```

What if we wanted to store this block, for execution later?

```ruby
{ puts "tweet" }
```

TWO WAYS 4 STORING BLOCKS

Proc.new

```ruby
my_proc = Proc.new { puts "tweet" }
my_proc.call  # => tweet
```

same as

```ruby
my_proc = Proc.new do
  puts "tweet"
end
my_proc.call  # => tweet
```

lambda

```ruby
my_proc = lambda { puts "tweet" }
my_proc.call  # => tweet
```

Ruby 1.9

```ruby
my_proc = -> { puts "tweet" }
my_proc.call  # => tweet
```

RUBY BITS

BLOCK TO LAMBDA

```ruby
class Tweet
  def post
    if authenticate?(@user, @password)
      # submit the tweet
      yield
    else
      raise 'Auth Error'
    end
  end
end
```

```ruby
class Tweet
  def post(success)
    if authenticate?(@user, @password)
      # submit the tweet
      success.call
    else
      raise 'Auth Error'
    end
  end
end
```

```ruby
tweet = Tweet.new('Ruby Bits!')
tweet.post { puts "Sent!" }
```

```ruby
tweet = Tweet.new('Ruby Bits!')
success = -> { puts "Sent!" }
tweet.post(success)
```


What about multiple lambdas?

MULTIPLE LAMBDAS

```ruby
class Tweet
  def post(success, error)
    if authenticate?(@user, @password)
      # submit the tweet
      success.call
    else
      error.call
    end
  end
end
```

```ruby
tweet = Tweet.new('Ruby Bits!')
success = -> { puts "Sent!" }
error = -> { raise 'Auth Error' }
tweet.post(success, error)
```


RUBY BITS

LAMBDA TO BLOCK

```ruby
tweets = ["First tweet", "Second tweet"]
tweets.each do |tweet|
  puts tweet
end
```

Lets try converting to a proc

```ruby
tweets = ["First tweet", "Second tweet"]
printer = lambda { |tweet| puts tweet }
tweets.each(printer)
```

```
ArgumentError: wrong number of
arguments (1 for 0)
```

each expects a block, not a proc

RUBY BITS

LAMBDA TO BLOCK

```ruby
tweets = ["First tweet", "Second tweet"]
tweets.each do |tweet|
  puts tweet
end
```

Lets try converting to a proc

```ruby
tweets = ["First tweet", "Second tweet"]
printer = lambda { |tweet| puts tweet }
tweets.each(&printer)
```

'&' turns proc into block

RUBY BITS

USING THE AMPERSAND

Calling a method with & in front of a parameter

```
tweets.each(&printer)
```

turns a proc into block

Defining a method with & in front of a parameter

```
def each(&block)
```

turns a block into a proc,
so it can be assigned to parameter

Often these are used together

PASSING BLOCKS THROUGH

```ruby
class Timeline
  attr_accessor :tweets

  def each
    tweets.each { |tweet| yield tweet }
  end
end
```

```ruby
timeline = Timeline.new(tweets)
timeline.each do |tweet|
  puts tweet
end
```

```ruby
class Timeline
  attr_accessor :tweets

  def each(&block)
    tweets.each(&block)
  end
end
```

- Block into proc
- proc back into a Block

RUBY BITS

SYMBOL#TO_PROC

```ruby
tweets.map { |tweet| tweet.user }
```

same thing

```ruby
tweets.map(&:user)
```

Uses the symbol as the method name to be called.

```ruby
tweets.map(&:user.name)
```

```
undefined method `name' for
:user:Symbol (NoMethodError)
```

Cannot do this!

RUBY BITS

OPTIONAL BLOCKS

```ruby
timeline = Timeline.new
timeline.tweets = ["One", "Two"]
```

Call print without block

```ruby
timeline.print    # => One, Two
```

Call print with block

```ruby
timeline.print { |tweet|
  "tweet: #{tweet}"
}
```

```
# => tweet: First
# => tweet: Second
```

```ruby
class Timeline
  attr_accessor :tweets

  def print
    if block_given?
      tweets.each { |tweet| puts yield tweet }
    else
      puts tweets.join(", ")
    end
  end
end
```

OPTIONAL BLOCKS

```ruby
class Tweet
  def initialize
    yield self if block_given?
  end
end
```

```ruby
Tweet.new do |tweet|
  tweet.status = "Set in initialize!"
  tweet.created_at = Time.now
end
```

can optionally pass in a block that receives a tweet object

RUBY BITS

CLOSURE

```ruby
def tweet_as(user)
  lambda { |tweet| puts "#{user}: #{tweet}" }
end
```

current state of local variables is preserved when a lambda is created

```ruby
gregg_tweet = tweet_as("greggpollack")
```

resolves to

```ruby
lambda { |tweet| puts "greggpollack: #{tweet}" }
```

```ruby
gregg_tweet.call("Mind blowing!")
# => greggpollack: Mind blowing!
```


RUBY
BITS
DYNAMIC CLASSES & METHODS
PRESS START

STRUCT

```ruby
class Tweet
  attr_accessor :user, :status

  def initialize(user, status)
    @user, @status = user, status
  end
end
```

Struct gives same functionality

```ruby
Tweet = Struct.new(:user, :status)
```

```ruby
tweet = Tweet.new('Gregg', 'compiling!')
tweet.user # => Gregg
tweet.status # => compiling!
```

STRUCT EXTRA METHODS

```ruby
class Tweet
  attr_accessor :user, :status

  def initialize(user, status)
    @user, @status = user, status
  end

  def to_s
    "#{user}: #{status}"
  end
end
```

```ruby
Tweet = Struct.new(:user, :status) do
  def to_s
    "#{user}: #{status}"
  end
end
```

ALIAS_METHOD

```ruby
class Timeline
  def initialize(tweets = [])
    @tweets = tweets
  end

  def tweets
    @tweets
  end

  def contents
    @tweets
  end
end
```

Same implementation but different names

RUBY BITS

ALIAS_METHOD

```ruby
class Timeline
  def initialize(tweets = [])
    @tweets = tweets
  end

  def tweets
    @tweets
  end

  alias_method :contents, :tweets
end
```

Makes 'contents' a new copy of the tweets method

RUBY BITS

ALIAS_METHOD

```ruby
class Timeline
  def initialize(tweets = [])
    @tweets = tweets
  end

  attr_reader :tweets

  alias_method :contents, :tweets
end
```

RUBY BITS

ALIAS_METHOD

```
class Timeline
  attr_accessor :tweets

  def print
    puts tweets.join("\n")
  end
end
```

previous version of the method can still be used

To add authentication outside the class

```
class Timeline
  alias_method :old_print, :print

  def print
    authenticate!
    old_print
  end

  def authenticate!
    # do some authentication here
  end
end
```

SUPER IS CLEANER

```ruby
class Timeline
  attr_accessor :tweets

  def print
    puts tweets.join("\n")
  end
end
```

```ruby
class AuthenticatedTimeline < Timeline
  def print
    authenticate!
    super
  end

  def authenticate!
    # do some authentication here
  end
end
```

DEFINE_METHOD

```ruby
class Tweet
  def draft
    @status = :draft
  end

  def posted
    @status = :posted
  end

  def deleted
    @status = :deleted
  end
end
```

```ruby
class Tweet
  states = [:draft, :posted, :deleted]
  states.each do |status|
    define_method status do
      @status = status
    end
  end
end
```

methods are created dynamically

SEND

```ruby
class Timeline
  def initialize(tweets)
    @tweets = tweets
  end

  def contents
    @tweets
  end

  private

  def direct_messages
  end
end
```

```ruby
tweets = ['Compiling!', 'Bundling...']
timeline = Timeline.new(tweets)
```

```ruby
timeline.contents
```

same

```ruby
timeline.send(:contents)
```

same

```ruby
timeline.send("contents")
```

```ruby
timeline.send(:direct_messages)
```

send can run private or protected methods

SEND

```ruby
class Timeline
  def initialize(tweets)
    @tweets = tweets
  end

  def contents
    @tweets
  end

  private

  def direct_messages
  end
end
```

```ruby
tweets = ['Compiling!', 'Bundling...']
timeline = Timeline.new(tweets)
```

```ruby
timeline.send(:direct_messages)
```

can run private or protected methods

```ruby
timeline.public_send("direct_messages")
```

```
private method `direct_messages' called for
#<Timeline:0x007fd273904eb0> (NoMethodError)
```

prevents running private or protected methods

RUBY BITS

THE METHOD METHOD

```ruby
class Timeline
  def initialize(tweets)
    @tweets = tweets
  end

  def contents
    @tweets
  end

  def show_tweet(index)
    puts @tweets[index]
  end
end
```

```ruby
tweets = ['Compiling!', 'Bundling...']
timeline = Timeline.new(tweets)
```

```ruby
content_method = timeline.method(:contents)
```

```
=> #<Method: Timeline#contents>
```

```ruby
content_method.call
```

```
=> ["Compiling!", "Bundling..."]
```

```ruby
show_method = timeline.method(:show_tweet)
```

```
=> #<Method: Timeline#show_tweet>
```

```ruby
show_method.call(0)
```

```
Compiling!
```

RUBY BITS

THE METHOD METHOD

```ruby
class Timeline
  def initialize(tweets)
    @tweets = tweets
  end

  def contents
    @tweets
  end

  def show_tweet(index)
    puts @tweets[index]
  end
end
```

```ruby
tweets = ['Compiling!', 'Bundling...']
timeline = Timeline.new(tweets)
```

```ruby
show_method = timeline.method(:show_tweet)
```

```
=> #<Method: Timeline#show_tweet>
```

```ruby
(0..1).each(&show_method)
```

same

```
Compiling!
Bundling...
```

turns the Method object into a Proc object

```ruby
show_method.call(0)
show_method.call(1)
```


RUBY
BITS
UNDERSTANDING & USING SELF
PRESS START

SELF

```
puts "Outside the class: #{self}"

class Tweet
  puts "Inside the class: #{self}"
end
```

```
Outside the class: main
Inside the class: Tweet
```

Ruby's special top level object

the Tweet class object

SELF

```ruby
class Tweet
  def self.find(keyword)
    puts "Inside a class method: #{self}"
  end
end

Tweet.find("rubybits")
```

```
Inside a class method: Tweet
```

the Tweet class object

RUBY BITS

CLASS METHODS

```ruby
class Tweet
  def self.find(keyword)
    # do stuff here
  end
end
```

```ruby
def Tweet.find(keyword)
  # do stuff here
end
```

this is equivalent, but not often used

SELF

```ruby
class Tweet
  def initialize(status)
    puts "Inside a method: #{self}"
    @status = status
  end
end

Tweet.new("What is self, anyway?")
```

the Tweet instance

```
Inside a method: #<Tweet:0x007f8c222de488>
```

RUBY BITS

FINDING METHODS

When there's no explicit receiver, look in self

```
class Tweet
  attr_accessor :status

  def initialize(status)
    @status = status
    set_up_some_things
  end

  def set_up_some_things
    # do something here
  end
end
```

called on the Tweet class object

instance variable added to the Tweet instance

called on the Tweet instance

RUBY BITS

CLASS_EVAL

Sets self to the given class and executes a block

```ruby
class Tweet
  attr_accessor :status, :created_at

  def initialize(status)
    @status = status
    @created_at = Time.now
  end
end
```

```ruby
Tweet.class_eval do
  attr_accessor :user
end
```

inside the block, self is the Tweet class

```ruby
tweet = Tweet.new("Learning class_eval with Ruby Bits")
tweet.user = "codeschool"
```

RUBY BITS

CREATING A METHOD LOGGER

```
class Tweet
  def say_hi
    puts "Hi"
  end
end

logger = MethodLogger.new
logger.log_method(Tweet, :say_hi)
```

tell the MethodLogger to log calls to say_hi

CREATING A METHOD LOGGER

How should MethodLogger work?

log_method takes a class and method name
use class_eval to execute code in the class
    use alias_method to save the original method
    use define_method to redefine the method
        log the method call
        use send to call the original method

CREATING A METHOD LOGGER

```ruby
class MethodLogger
  def log_method(klass, method_name)
    klass.class_eval do
      alias_method "#{method_name}_original", method_name
      define_method method_name do
        puts "#{Time.now}: Called #{method_name}"
        send "#{method_name}_original"
      end
    end
  end
end
```

save the original method

log the method call

call the original method

CREATING A METHOD LOGGER

```
class Tweet
  def say_hi
    puts "Hi"
  end
end

logger = MethodLogger.new
logger.log_method(Tweet, :say_hi)

Tweet.new.say_hi
```

```
2012-09-01 12:52:03 -400: Called say_hi
Hi
```

CREATING A METHOD LOGGER

Bonus: Log methods with params and blocks

```
class MethodLogger
  def log_method(klass, method_name)
    klass.class_eval do
      alias_method "#{method_name}_original", method_name
      define_method method_name do |*args, &block|
        puts "#{Time.now}: Called #{method_name}"
        send "#{method_name}_original", *args, &block
      end
    end
  end
end
```

capture args and block

pass them to the original method

RUBY BITS

INSTANCE_EVAL

Sets self to the given instance and executes a block

```ruby
class Tweet
  attr_accessor :user, :status
end
```

```ruby
tweet = Tweet.new
tweet.instance_eval do
  self.status = "Changing the tweet's status"
end
```

....inside the block,
self is the Tweet instance

RUBY BITS

INSTANCE_EVAL

```ruby
class Tweet
  attr_accessor :user, :status

  def initialize
    yield self if block_given?
  end
end

Tweet.new do |tweet|
  tweet.status = "I was set in the initialize block!"
  tweet.user = "Gregg"
end
```

... our block needs the tweet instance

can we clean this up?

INSTANCE_EVAL

```ruby
class Tweet
  attr_accessor :user, :status

  def initialize(&block)
    instance_eval(&block) if block_given?
  end
end

Tweet.new do
  self.status = "I was set in the initialize block!"
  self.user = "Gregg"
end
```

capture the block

pass it to instance_eval

RUBY BITS


RUBY
BITS
HANDLING MISSING METHODS
PRESS START

METHOD_MISSING

Called when Ruby can't find a method

```ruby
class Tweet
  def method_missing(method_name, *args)
    puts "You tried to call #{method_name} with these arguments: #{args}"
  end
end

Tweet.new.submit(1, "Here's a tweet.")
```

```
You tried to call submit with arguments: [1, "Here's a tweet."]
```

RUBY BITS

METHOD_MISSING

```ruby
class Tweet
  def method_missing(method_name, *args)
    logger.warn "You tried to call #{method_name} with these arguments: #{args}"
    super
  end
end

Tweet.new.submit(1, "Here's a tweet.")
```

write to our log

Ruby's default method_missing handling raises a NoMethodError

DELEGATING METHODS

```ruby
class Tweet
  def initialize(user)
    @user = user
  end

  def username
    @user.username
  end

  def avatar
    @user.avatar
  end
end
```

starting to see duplication,
and what if we need to add more of these?

DELEGATING METHODS

```
class Tweet
  def initialize(user)
    @user = user
  end

  def method_missing(method_name, *args)
    @user.send(method_name, *args)
  end
end
```

send all unknown method calls to the user

RUBY BITS

DELEGATING METHODS

```ruby
class Tweet
  DELEGATED_METHODS = [:username, :avatar]

  def initialize(user)
    @user = user
  end

  def method_missing(method_name, *args)
    if DELEGATED_METHODS.include?(method_name)
      @user.send(method_name, *args)
    else
      super
    end
  end
end
```

delegate only certain methods

default handling for all other methods

RUBY BITS

SIMPLE DELEGATOR

```ruby
require 'delegate'

class Tweet < SimpleDelegator
  def initialize(user)
    super(user)
  end
end
```

automatically delegates all unknown methods to user

DYNAMIC METHODS

How can we add this functionality with method_missing?

```
tweet = Tweet.new("Sponsored by")
tweet.hash_ruby
tweet.hash_metaprogramming
puts tweet
```

```
Sponsored by #ruby #metaprogramming
```

We can call any method with hash_*

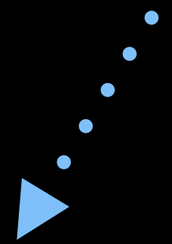
DYNAMIC METHODS

```ruby
class Tweet
  def initialize(text)
    @text = text
  end

  def to_s
    @text
  end

  def method_missing(method_name, *args)
    match = method_name.to_s.match(/^hash_(\w+)/)
    if match
      @text << " #" + match[1]
    else
      super
    end
  end
end
```

```ruby
tweet = Tweet.new("Sponsored by")
tweet.hash_ruby
tweet.hash_metaprogramming
puts tweet
```



RUBY BITS

RESPOND_TO?

Tells us if an object responds to a given method

```
tweet = Tweet.new
tweet.respond_to?(:to_s)       # => true
tweet.hash_ruby
tweet.respond_to?(:hash_ruby) # => false
```

... works because we defined method_missing

lies!!!

RUBY BITS

RESPOND_TO?

```ruby
class Tweet
  ...
  def respond_to?(method_name)
    method_name =~ /^hash_\w+/ || super
  end
end
```

respond to methods starting with hash_

default handling for everything else

```ruby
tweet = Tweet.new
tweet.respond_to?(:hash_ruby)  # => true
```

```ruby
tweet.method(:hash_ruby)
```

NameError: undefined method

RUBY BITS

RESPOND_TO_MISSING?

Ruby 1.9.3

```
class Tweet
  ...
  def respond_to_missing?(method_name)
    method_name =~ /^hash_\w+/ || super
  end
end
```

```
tweet = Tweet.new
tweet.method(:hash_ruby)
```

returns a Method object as expected

DEFINE_METHOD REVISITED

```ruby
def method_missing(method_name, *args)
  match = method_name.to_s.match(/^hash_(\w+)/)
  if match
    @text << " #" + match[1]
  else
    super
  end
end
```

```ruby
tweet.hash_codeschool
tweet.hash_codeschool
```

calls method_missing both times

RUBY BiTS

DEFINE_METHOD REVISITED

```ruby
def method_missing(method_name, *args)
  match = method_name.to_s.match(/^hash_(\w+)/)
  if match
    self.class.class_eval do
      define_method(method_name) do
        @text << " #" + match[1]
      end
    end
    send(method_name)
  else
    super
  end
end
```

execute in context of the class

define a new method

then call it

```ruby
def hash_codeschool
  @text << " #" + "codeschool"
end
```

```ruby
tweet.hash_codeschool
tweet.hash_codeschool
```

calls method_missing

calls hash_codeschool

RUBY BITS


RUBY
BITS
DSL PART 1
PRESS START

DOMAIN SPECIFIC LANGUAGES

DSL a language that has terminology and constructs designed for a specific domain

External DSL a standalone DSL

Internal DSL a DSL implemented within another programming language

RUBY BITS

OUR SAMPLE DSL

```
tweet_as 'markkendall' do
  mention 'codeschool'
  text 'I made a DSL!'
  hashtag 'hooray'
  hashtag 'ruby'
  link 'http://codeschool.com'
end
```

valid Ruby syntax with custom terminology

```
@codeschool I made a DSL! #hooray #ruby http://codeschool.com
```

RUBY BITS

FIRST IMPLEMENTATION

```ruby
tweet_as 'markkendall' do
  mention 'codeschool'
  text 'I made a DSL!'
  hashtag 'hooray'
  hashtag 'ruby'
  link 'http://codeschool.com'
end
```

```ruby
def tweet_as(user)
  @user = user
  @tweet = []
  yield
  submit_to_twitter
end
```

```ruby
def text(str)
  @tweet << str
end

def mention(user)
  @tweet << "@" + user
end

def hashtag(str)
  @tweet << "#" + str
end

def link(str)
  @tweet << str
end
```

RUBY BITS

FIRST IMPLEMENTATION

```
tweet_as 'markkendall' do
  mention 'codeschool'
  text 'I made a DSL!'
  hashtag 'hooray'
  hashtag 'ruby'
  link 'http://codeschool.com'
end
```

We polluted our namespace!!

```
def tweet_as(user)
  @user = user
  @tweet = []
  yield
  submit_to_twitter
end

def submit_to_twitter
  tweet_text = @tweet.join(' ')
  puts "#{@user}: #{tweet_text}"
end

def text(str)
  @tweet << str
end
...
```

NAMESPACE POLLUTION

```ruby
def tweet_as(user)
  tweet = Tweet.new(user)
  yield tweet
  tweet.submit_to_twitter
end
```

new block parameter

implementation didn't change

```ruby
class Tweet
  def initialize(user)
    @user = user
    @tweet = []
  end

  def submit_to_twitter
    tweet_text = @tweet.join(' ')
    puts "#{@user}: #{tweet_text}"
  end

  def text(str)
    @tweet << str
  end
  ...
end
```

NAMESPACE POLLUTION

```ruby
def tweet_as(user)
  tweet = Tweet.new(user)
  yield tweet
  tweet.submit_to_twitter
end
```

```ruby
tweet_as 'markkendall' do |tweet|
  tweet.mention 'codeschool'
  tweet.text 'I made a DSL!'
  tweet.hashtag 'hooray'
  tweet.hashtag 'ruby'
  tweet.link 'http://codeschool.com'
end
```

we made our DSL syntax ugly

RUBY BITS

INSTANCE_EVAL

```ruby
def tweet_as(user, &block)
  tweet = Tweet.new(user)
  tweet.instance_eval(&block)
  tweet.submit_to_twitter
end
```

capture the block,
pass it to instance_eval

```ruby
tweet_as 'markkendall' do
  mention 'codeschool'
  text 'I made a DSL!'
  hashtag 'hooray'
  hashtag 'ruby'
  link 'http://codeschool.com'
end
```

our original, clean syntax is back!

RUBY BITS

METHOD CHAINING

```ruby
tweet_as 'markkendall' do
  mention 'codeschool'
  text('I made a DSL!').hashtag('hooray').hashtag('ruby')
  link 'http://codeschool.com'
end
```

Ruby requires these parens

```ruby
def text(str)
  @tweet << str
  self
end
```

return self

```ruby
def mention(user)
  @tweet << "@" + user
  self
end
```

```ruby
def hashtag(str)
  @tweet << "#" + str
  self
end
```

RUBY BITS


RUBY
BITS
DSL PART 2
PRESS START

SPECIAL CASES

Tweet with just text

```
tweet_as 'markkendall' do
  text 'This is a simple tweet'
end
```

```
tweet_as 'markkendall', 'This is a simple tweet'
```

this looks more natural

RUBY BITS

SPECIAL CASES

```
tweet_as 'markkendall', 'This is a simple tweet'
```

```
def tweet_as(user, &block)
  tweet = Tweet.new(user)
  tweet.instance_eval(&block)
  tweet.submit_to_twitter
end
```

old method

RUBY BITS

SPECIAL CASES

```
tweet_as 'markkendall', 'This is a simple tweet'
```

```
def tweet_as(user, text = nil, &block)
  tweet = Tweet.new(user)
  tweet.text(text) if text
  tweet.instance_eval(&block) if block_given?
  tweet.submit_to_twitter
end
```

optional text

optional block

RUBY BITS

SPECIAL CASES

```
tweet_as 'markkendall' do
  mention 'codeschool'
  text('I made a DSL!').hashtag('hooray').hashtag('ruby')
  link 'http://codeschool.com'
end
```

```
markkendall: @codeschool I made a DSL! #hooray #ruby http://codeschool.com
```

```
tweet_as 'markkendall', 'This is a simple tweet'
```

```
markkendall: This is a simple tweet
```

RUBY BITS

ARRAY PARAMETERS

Let's allow for multiple mentions

```
tweet_as 'markkendall' do
  mention 'codeschool'
  mention 'envylabs'
  text 'I made a DSL!'
end
```

```
tweet_as 'markkendall' do
  mention 'codeschool', 'envylabs'
  text 'I made a DSL!'
end
```

RUBY BITS

ARRAY PARAMETERS

```
tweet_as 'markkendall' do
  mention 'codeschool', 'envylabs'
  text 'I made a DSL!'
end
```

```
def mention(user)
  @tweet << "@" + user
  self
end
```

old code

we want an array here

RUBY BITS

ARRAY PARAMETERS

```
tweet_as 'markkendall' do
  mention 'codeschool', 'envylabs'
  text 'I made a DSL!'
end
```

```
def mention(*users)
  users.each do |user|
    @tweet << "@" + user
  end
  self
end
```

puts all the parameters into an array

RUBY BITS

DYNAMIC METHODS

Let's add support for tweet annotations

```
tweet_as 'markkendall' do
  mention 'codeschool', 'envylabs'
  text 'I made a DSL!'
  hashtag 'hooray'
  hashtag 'ruby'
  link 'http://codeschool.com
  lat 28.415833
  lng -81.298889
  keywords 'Ruby', 'DSL'
end
```

creates a hash:

```
{
  lat: 28.415833,
  lng: -81.298889,
  keywords: "Ruby, DSL"
}
```

RUBY BITS

DYNAMIC METHODS

old code

```
class Tweet
  def initialize(user)
    @user = user
    @tweet = []
  end

  def submit_to_twitter
    tweet_text = @tweet.join(' ')
    puts "#{@user}: #{tweet_text}"
  end
end
```

RUBY BITS

DYNAMIC METHODS

```ruby
class Tweet
  def initialize(user)
    @user = user
    @tweet = []
    @annotations = {}
  end

  def submit_to_twitter
    tweet_text = @tweet.join(' ')
    puts "#{@user}: #{tweet_text}"
    puts @annotations.inspect unless @annotations.empty?
  end

  def method_missing(method, *args)
    @annotations[method] = args.join(", ")
  end
end
```

method params become hash values

unknown method names become hash keys

RUBY BITS

DYNAMIC METHODS

```
tweet_as 'markkendall' do
  mention 'codeschool', 'envylabs'
  text 'I made a DSL!'
  hashtag 'hooray'
  hashtag 'ruby'
  link 'http://codeschool.com
  lat 28.415833
  lng -81.298889
  keywords 'Ruby', 'DSL'
end
```

```
markkendall: @codeschool @envylabs I made a DSL! #hooray #ruby http://codeschool.com
{:lat=>"28.415833", :lng=>"-81.298889", :keywords=>"Ruby, DSL"}
```

RUBY BITS

ERROR HANDLING

Let's check to make sure the tweet isn't too long

old code

```
def submit_to_twitter
  tweet_text = @tweet.join(' ')
  puts "#{@user}: #{tweet_text}"
  puts @annotations.inspect unless @annotations.empty?
end
```

ERROR HANDLING

```ruby
def submit_to_twitter
  tweet_text = @tweet.join(' ')
  if tweet_text.length <= 140
    puts "#{@user}: #{tweet_text}"
    puts @annotations.inspect unless @annotations.empty?
  else
    raise "Your tweet is too long."
  end
end
```

raise an exception
with a descriptive message

ERROR HANDLING

```
tweet_as 'markkendall' do
  text "This tweet is way too long and Twitter won't know how to handle it \
  because Twitter imposes a limit on the number of characters in a tweet and \
  this is a run-on sentence."
  hashtag 'lol'
  hashtag 'twitter'
end
```

```
in `submit_to_twitter': Your tweet is too long. (RuntimeError)
```