

RUBY



BLOCKS, PROCS, AND LAMBDRAS

PRESS START

REMEMBER THIS?

```
def call_this_block_twice  
  yield  
  yield  
end
```

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

..... ➤ tweet
 tweet

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

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

RUBY
BITS

TWO WAYS 4 STORING BLOCKS

Proc.new

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

same as

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

lambda

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

Ruby 1.9

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

RUBY
BITS

BLOCK TO LAMBDA

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

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

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

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

MULTIPLE LAMBDAS

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

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

LAMBDA TO BLOCK

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

↳ Lets try converting to a proc

```
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

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

• Let's try converting to a proc

```
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

```
class Timeline
  attr_accessor :tweets

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



```
class Timeline
  attr_accessor :tweets

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



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

- Block into proc
- proc back into a Block

RUBY
BITS

SYMBOL#TO_PROC

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

same thing



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

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

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

Cannot do this!

Uses the symbol as the method name to be called.

RUBY
BITS

OPTIONAL BLOCKS

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

Call print without block

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

Call print with block

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

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

```
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

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

```
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

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

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

```
gregg_tweet = tweet_as("greggpollack") ••• resolves to
          ↓
lambda { |tweet| puts "greggpollack: #{tweet}" }
```

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

RUBY
BITS

RUBY



DYNAMIC CLASSES & METHODS

PRESS START

STRUCT

```
class Tweet
  attr_accessor :user, :status
  def initialize(user, status)
    @user, @status = user, status
  end
end
```

Struct gives same functionality

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

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

RUBY
BITS

STRUCT EXTRA METHODS

```
class Tweet
  attr_accessor :user, :status
  def initialize(user, status)
    @user, @status = user, status
  end

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

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

ALIAS_METHOD

```
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

```
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

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

  attr_reader :tweets

  alias_method :contents, :tweets
end
```



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
```

RUBY
BITS

SUPER IS CLEANER

```
class Timeline
attr_accessor :tweets

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

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

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



DEFINE_METHOD

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

  def posted
    @status = :posted
  end

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

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

methods are created dynamically

RUBY
BITS

SEND

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

  def contents
    @tweets
  end

  private

  def direct_messages
  end
end
```

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

```
timeline.contents
```

: same

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

: same

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

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

send can run private
or protected methods

RUBY
BITS

SEND

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

  def contents
    @tweets
  end

  private

  def direct_messages
  end
end
```

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

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

can run private or protected methods

```
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

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

  def contents
    @tweets
  end

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

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

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

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

```
content_method.call
```

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

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

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

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

```
Compiling!
```



THE METHOD METHOD

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

  def contents
    @tweets
  end

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

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

```
show_method = timeline.method(:show_tweet)
=> #<Method: Timeline#show_tweet>
```

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

Same

Compiling! turns the Method object
Bundling... into a Proc object

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

RUBY
BITS

RUBY



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

RUBY
BITS

SELF

```
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

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

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

this is equivalent, but not often used

RUBY
BITS

SELF

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

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

Inside a method: #<Tweet:0x007f8c222de488>

...the Tweet instance

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

```
class Tweet
  attr_accessor :status, :created_at

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

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

inside the block,
`self` is the `Tweet` class

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



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

RUBY
BITS

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

```
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}" <---- log the method call
        send "#{method_name}_original"
      end
    end
  end
end
```

• save the original method

• log the method call

• call the original method

RUBY
BITS

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
```

RUBY
BITS

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

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

```
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

```
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?

RUBY
BITS

INSTANCE_EVAL

```
class Tweet
  attr_accessor :user, :status

  def initialize(&block) <----- capture the 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
```

..... pass it to
instance_eval

RUBY
BITS

RUBY



HANDLING MISSING METHODS

PRESS START

METHOD_MISSING

Called when Ruby can't find a method

```
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."]



METHOD_MISSING

```
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

```
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?

RUBY
BITS

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

```
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

```
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

```
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
```

```
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?

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

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

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

NameError: undefined method

respond to methods
starting with hash_
default handling for
everything else



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_METHODO0 REVISITED

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

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

calls method_missing both times

RUBY
BITS

DEFINE_METHODO0 REVISITED

```
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 define a new method
        @text << "#" + match[1]
      end
    end
    send(method_name) then call it
  else
    super
  end
end
```

execute in context
of the class

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

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

calls method_missing :
calls hash_codeschool

RUBBY



D S L P A R T 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



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
```

FIRST IMPLEMENTATION

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

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

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

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

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

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

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
...
```

RUBY
BITS

NAMESPACE POLLUTION

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

new block parameter

implementation didn't change

```
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
```



JBY
BITS

NAMESPACE POLLUTION

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

```
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

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

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



..... capture the block,
..... pass it to instance_eval

our original, clean syntax
is back!

RUBY
BITS

METHOD CHAINING

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

• Ruby requires these parens

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

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

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

• return self

RUBBY



D^XL 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
```



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"
```

DYNAMIC METHODS

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

old code

RUBY
BITS

DYNAMIC METHODS

```
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"}
```



ERROR HANDLING

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

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

old code



ERROR HANDLING

```
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

RUBY
BITS

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)

