But the Language Supports it

Good morning everyone and thank you for being here.

aakash
dharmadhikari

aakashd

I do my open source here

@aakashd

Albeit intermittently I tweet here

niranjan paranjape

This is Niranjan

achamian

you can find his open source contribution here

@niranjan_p

he tweets here

C42 Engineering

http://github.com/c42

GOTO :context

design discussion

criticising
discuss design or architecture
trivial part of the system or whole codebase

If there is one common thing that we all love, it’s criticising code. We discuss the design or if you excuse me “the architecture” of the system we are working on. It may be a trivial part of the system or it might impact the whole codebase. The fact is we like to talk.

debate

Some time these discussions lead to a debate

rational arguments

To begin with rational arguments are presented by everyone involved to justify why certain thing should or should not be done. Points are put forward, counter-points are offered and merits and demerits of various approaches are taken into consideration. This often leads to a conclusion and everyone gets back to coding. But unfortunately sometimes this doesn't happen.

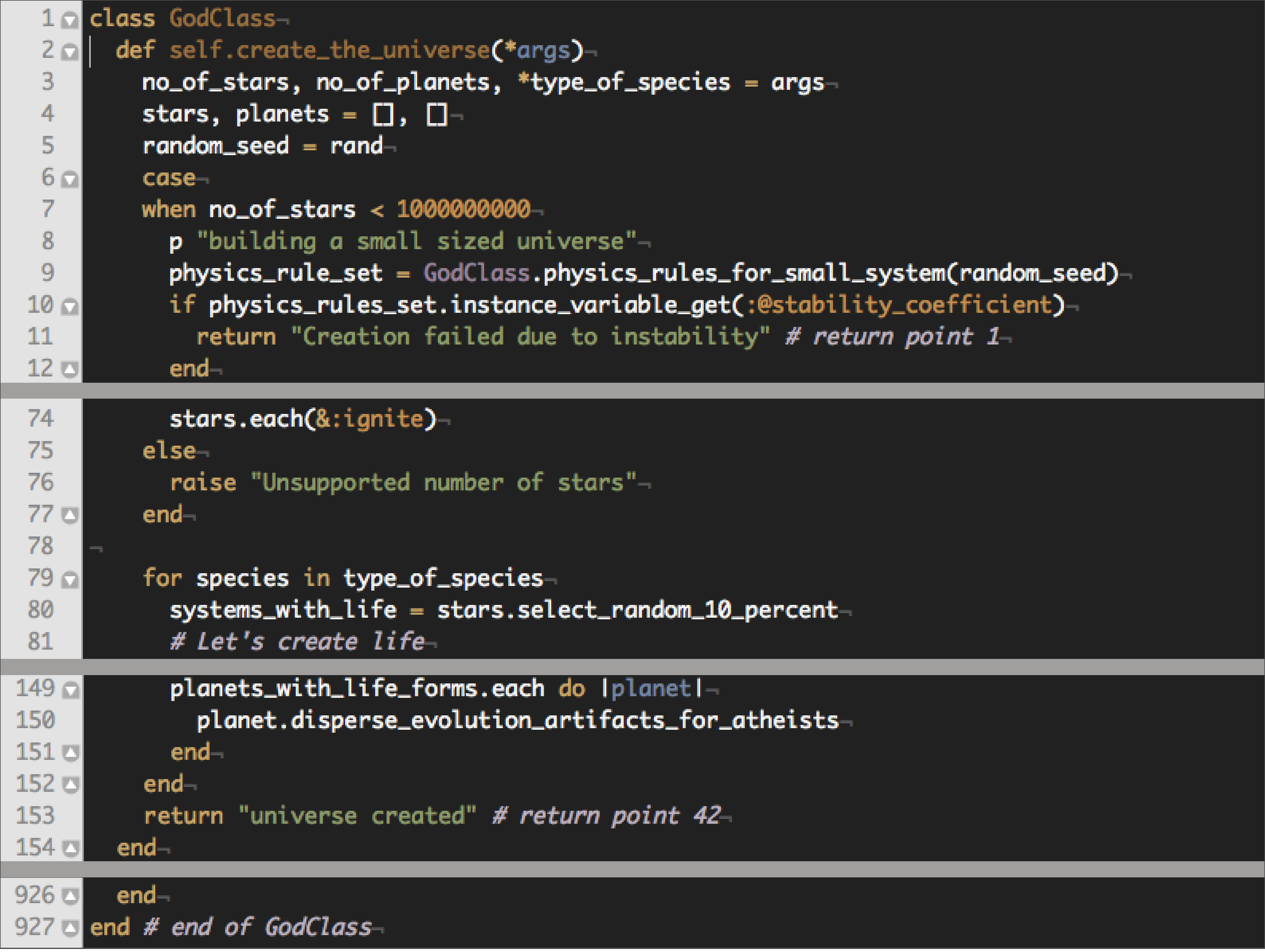
defensive arguments

Despite of all the odds, someone decides to stick with his solution and starts offering defensive arguments. It’s fine, we can agree to disagree.

But then there comes a time when the unspeakable is spoken.

but the language supports it

The language designers weren't stupid, why can't I use this feature when the language supports it really?



```ruby
1   class GodClass
2     def self.create_the_universe(*args)
3       no_of_stars, no_of_planets, *type_of_species = args
4       stars, planets = [], []
5       random_seed = rand
6       case
7       when no_of_stars < 1000000000
8         p "building a small sized universe"
9         physics_rule_set = GodClass.physics_rules_for_small_system(random_seed)
10        if physics_rules_set.instance_variable_get(:@stability_coefficient)
11          return "Creation failed due to instability" # return point 1
12        end
74        stars.each(&:ignite)
75      else
76        raise "Unsupported number of stars"
77      end
78
79      for species in type_of_species
80        systems_with_life = stars.select_random_10_percent
81        # Let's create life
149       planets_with_life_forms.each do |planet|
150         planet.disperse_evolution_artifacts_for_atheists
151       end
152     end
153     return "universe created" # return point 42
154   end
926   end
927 end # end of GodClass
```

Yes indeed, the language does support it. Language also supports writing 900 odd lines long class with 150 lines long method. Which has a switch case, for loop, 42 return statements, a few raise conditions and opening up a class like Array to add domain logic.

All these things and many more are supported by Ruby. There are places where you’ll use these features. But you shouldn’t treat everything as nail just because you are holding a hammer.

OO/Ruby

In next 30 minutes we will talk about various things which Ruby supports, few are specific to Ruby, while other are more generic in nature. We will talk about why you need to pay attention whenever you use any of these features.

In this talk we will cover various features provided by Ruby that, while awesome, should be used with caution. Few of these features are specific to Ruby, few are language agnostic.

library vs. domain

Most of these constructs might seem like a norm and necessary if you have been writing lot of library code to provide a generic feature set, but while writing domain specific (production) code they can be avoided to achieve better object oriented design

mechanic

One thing I'ld like to point out is, most of this is a mechanic to achieve a better code.

maturity of the team

[Handoff]

Depending on the maturity of team, you may use few of these construct without adding ‘Magic’ for the fellow programmer, while you may stay away from others by instinct.

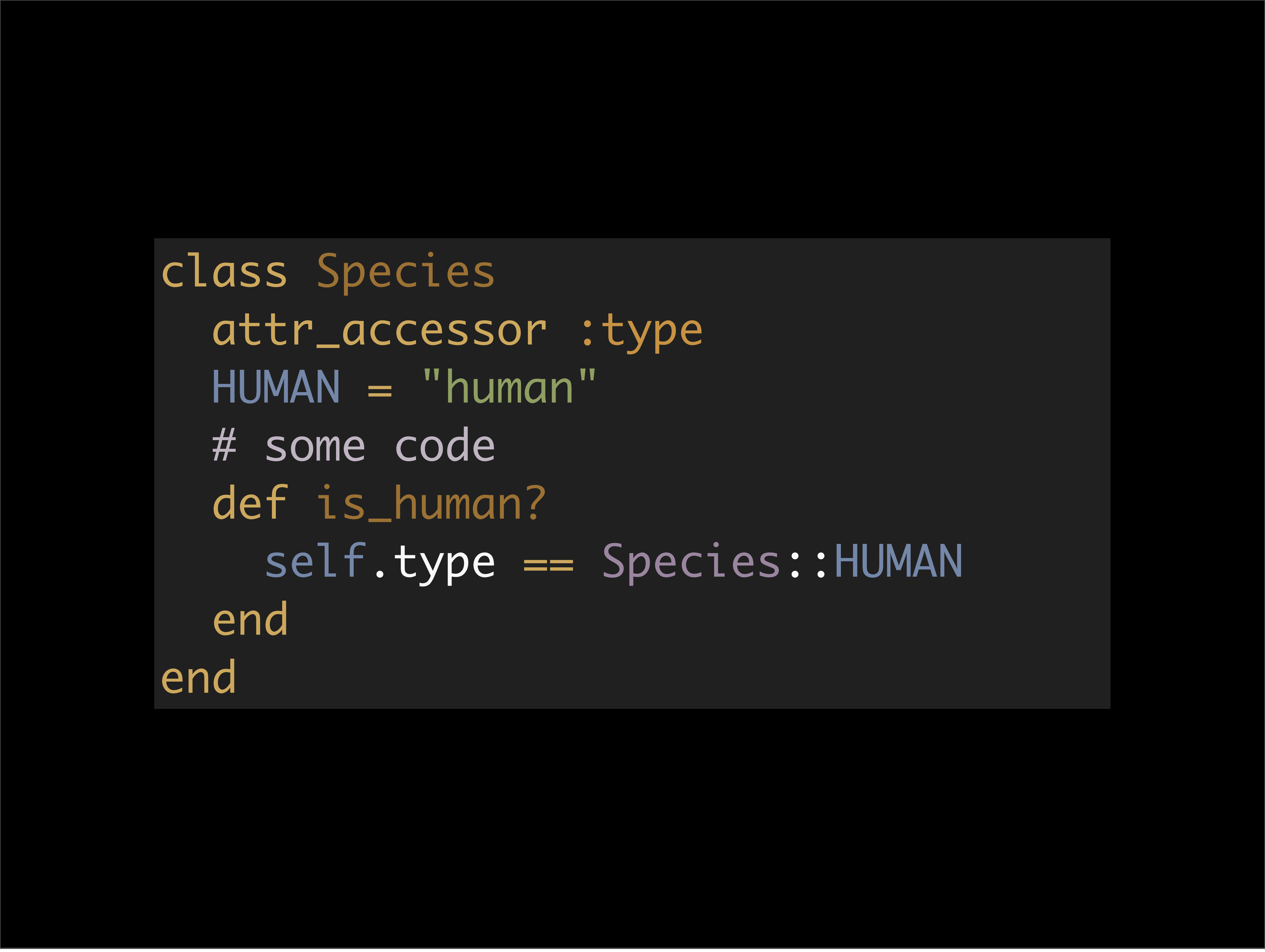
When I say fellow developers, I am not talking about your current team but also the developers who are going to inherit your code.

GOTO :begin

raise "it should have jumped to the next topic"

[achamian]

I guess that's enough of context let's skip through rest of the context and jump directly to the crux of the talk



```ruby
class Species
  attr_accessor :type
  HUMAN = "human"
  # some code
  def is_human?
    self.type == Species::HUMAN
  end
end
```

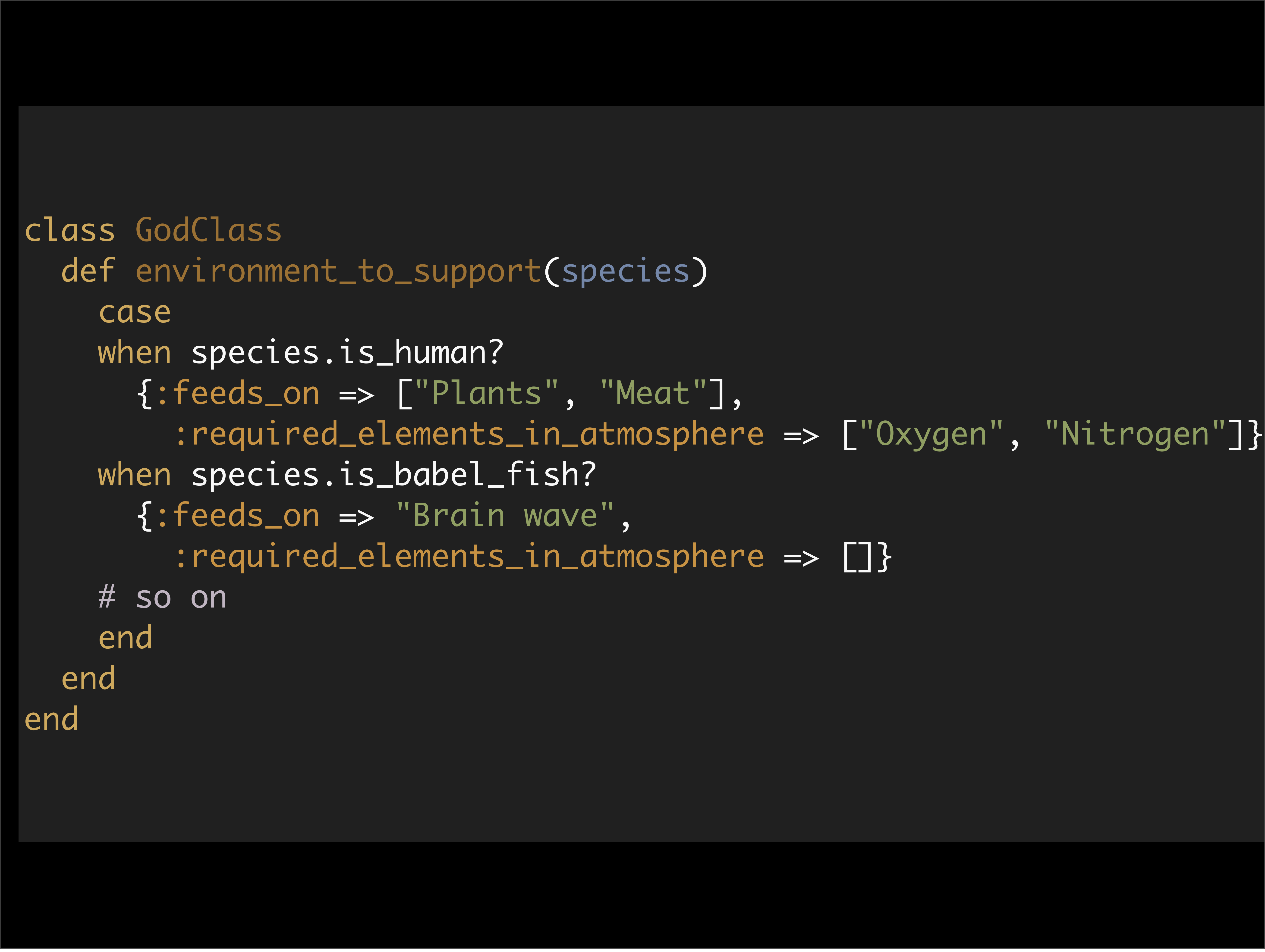
Jumping straight into the code example.

What's wrong with this method:

anyone?

It's as short as a method can get.
Name expresses the intent.
It returns a boolean, appropriately indicated by "?" at the end as per the Ruby convention

Looks perfect and it is. But it indicates a smell. Every time I write a method like this, I ask why anyone needs to know what state my object is in. The very existence of this method indicates that there's an "if" or worse "switch" somewhere else in my code. May be something like this


```
class GodClass
  def environment_to_support(species)
    case
    when species.is_human?
      {:feeds_on => ["Plants", "Meat"],
        :required_elements_in_atmosphere => ["Oxygen", "Nitrogen"]}
    when species.is_babel_fish?
      {:feeds_on => "Brain wave",
        :required_elements_in_atmosphere => []}
    # so on
    end
  end
end
```

Sweet, we have a switch case.

Nice readable code, is it testable? Who cares?

i love switch

Now that’s a problem our small, beautiful method created for us.

tell don’t ask

We are violating one of the basic principles of OO “tell don’t ask”. How to fix it, maybe it’s time to create a different classes for Human and BabelFish. May be something like

```
class BabelFish < Species
  def environmental_necessities
    {:feeds_on => "Brain wave",
      :required_elements_in_atmosphere => []}
  end
end

class GodClass
  def environment_to_support(species)
    species.environmental_necessities
  end
end
```

are “?” methods evil?

Not always. There are places where you have to make a choice between either breaking “tell don’t ask” or pushing code to an objects where it doesn’t belong. In such cases you might be forced to expose internal state of the object to outside world. For example:

```ruby
class User
  def is_admin?
    self.role == :admin
  end
end

class ApplicationController < AbstractController::Base
  def require_admin_privilege
    unless current_user.is_admin?
      head :unauthorized
    end
  end
end
```

Yes, this can be pushed into user by passing a lambda etc. but at least it is acceptable as you don't want to push controller's responsibility to a model

Back to switch

should ‘switch’ be banned?

While most of the places switch can be replaced with a better modelling, there are scenarios where it’s difficult to avoid switch. The only example I can think of is parsing an input from a command line so can I replace it with

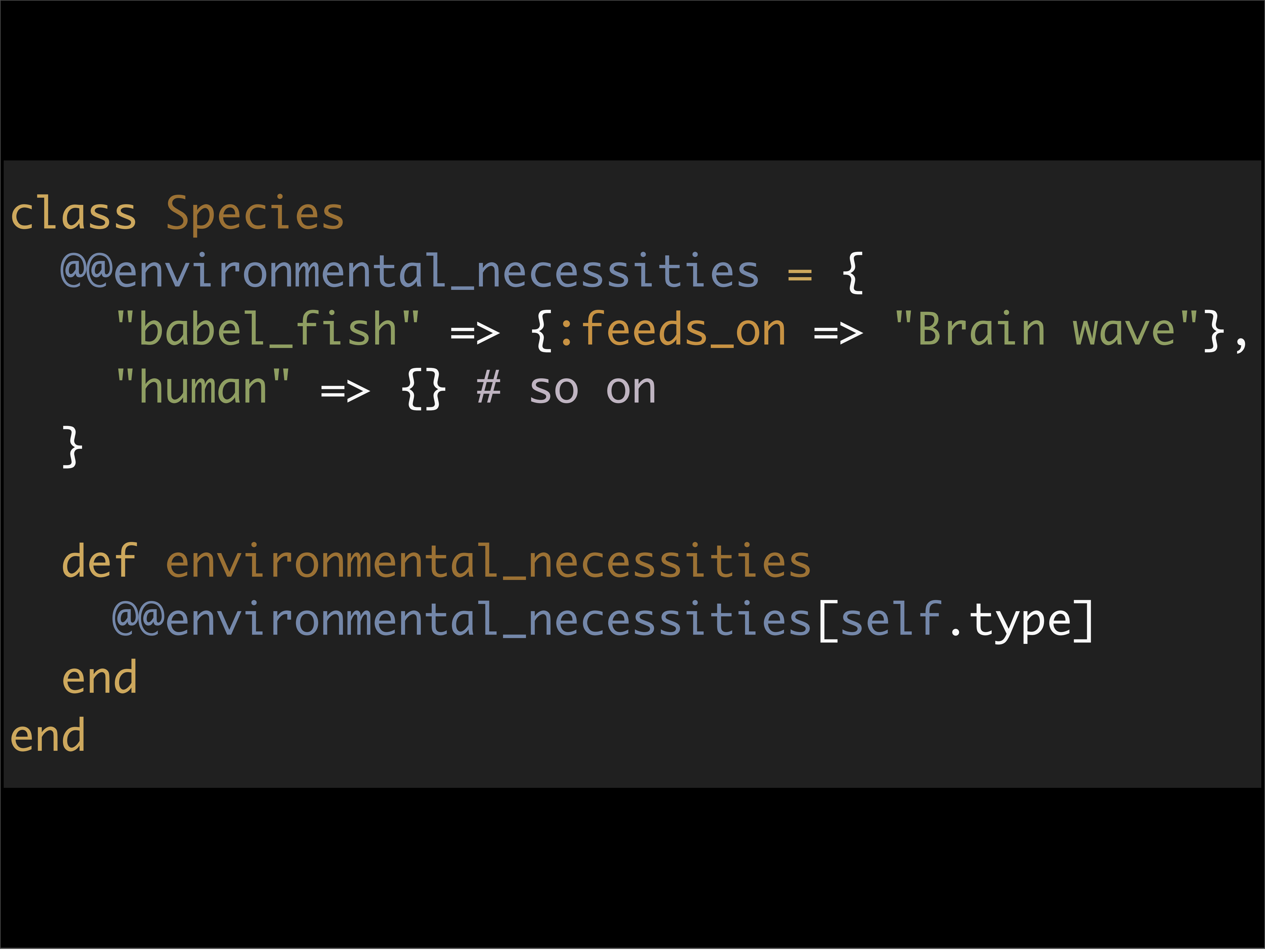
if elsif elsif elsif ...

http://www.flickr.com/photos/santiagogprg/5023648200/

Hmm.

http://www.flickr.com/photos/santiagogprg/5023648200/

how about this?



```ruby
class Species
  @@environmental_necessities = {
    "babel_fish" => {:feeds_on => "Brain wave"},
    "human" => {} # so on
  }

  def environmental_necessities
    @@environmental_necessities[self.type]
  end
end
```

[Handoff]

No switch, no if, no exposing the type.

The branching still exists.

While all that is true I'ld rather leave my code with an if or a switch and make the branching explicit so as to refactor it later instead of hiding the branching behind a Hash.

This is strictly a personal preference.

Raise

[aakashd]

What I want to talk about raising & rescuing exceptions is based on an assumption that we all know about two possible program flows; the regular flow & exceptional flow. That one should not throw an exception in regular flow & should use it carefully even in exceptional flow.

```ruby
class Aircraft
  def start_engine
    # try to start the engine
    raise "Unable to start engine" unless @engine.started?
  end
end
```

```ruby
class Aircraft
  def deploy_landing_gear
    # try to deploy the landing gear
    return @landing_gear.deployed?
  end
end
```

```ruby
class Aircraft
  def deploy_landing_gear
    # try to deploy the landing gear

    # landing gear is stuck halfway through
    raise "Prepare for crash landing" if @landing_gear.stuck?

    return @landing_gear.deployed?
  end
end
```

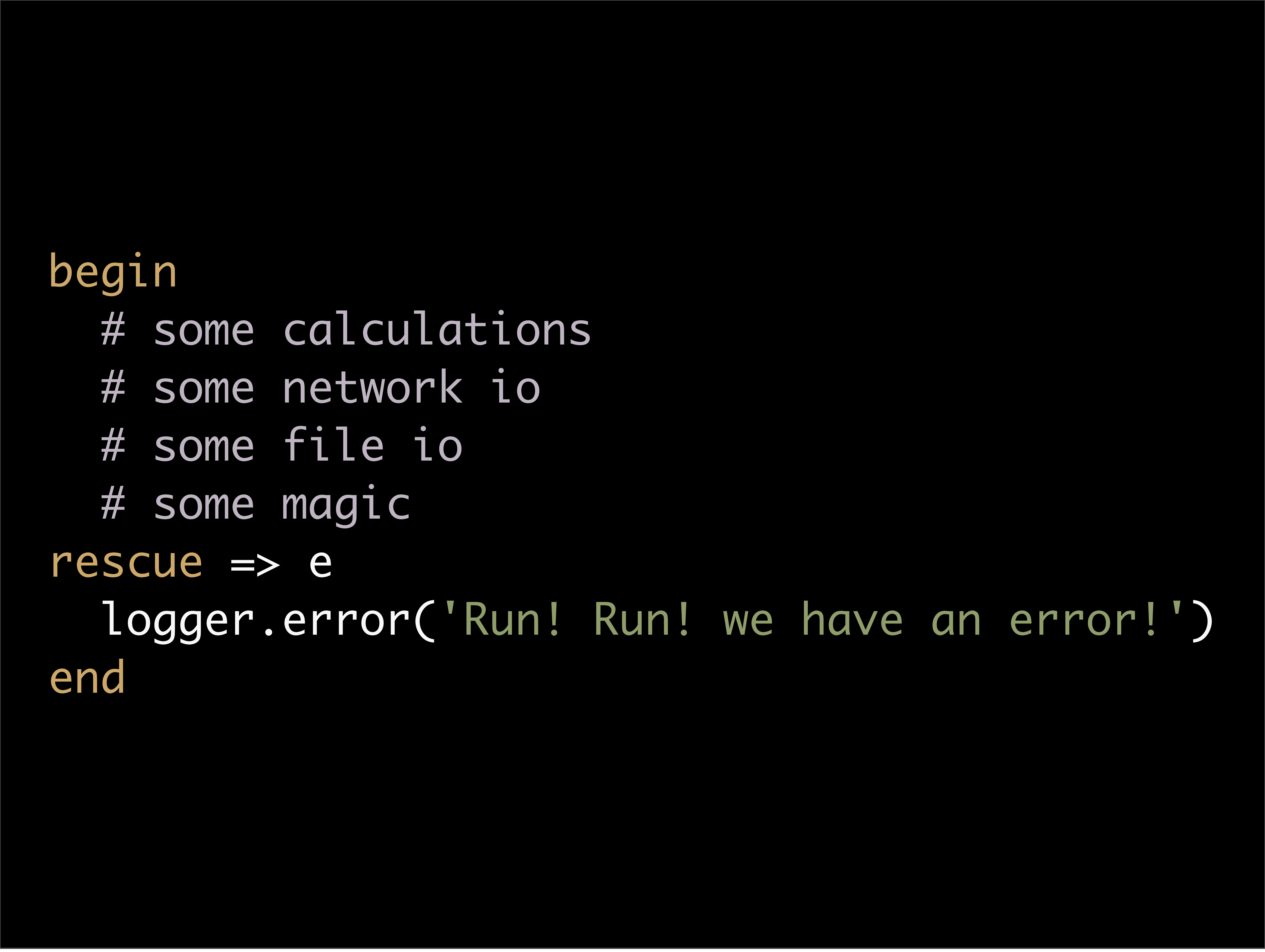
But what happens when landing gear is stuck.

rules to raise

One should raise an exception when the purpose of the method cannot be completed and because of that the application is left in an unknown and possible inconsistent state.

In languages like Java, exceptions are way to force user to handle a possible outcome of method invocation, which I believe is very much useful because hardly any of the developers take pain to read the documentation.

And last but not the least is the teams or community's conventions about using exceptions. If nothing please stick to that.

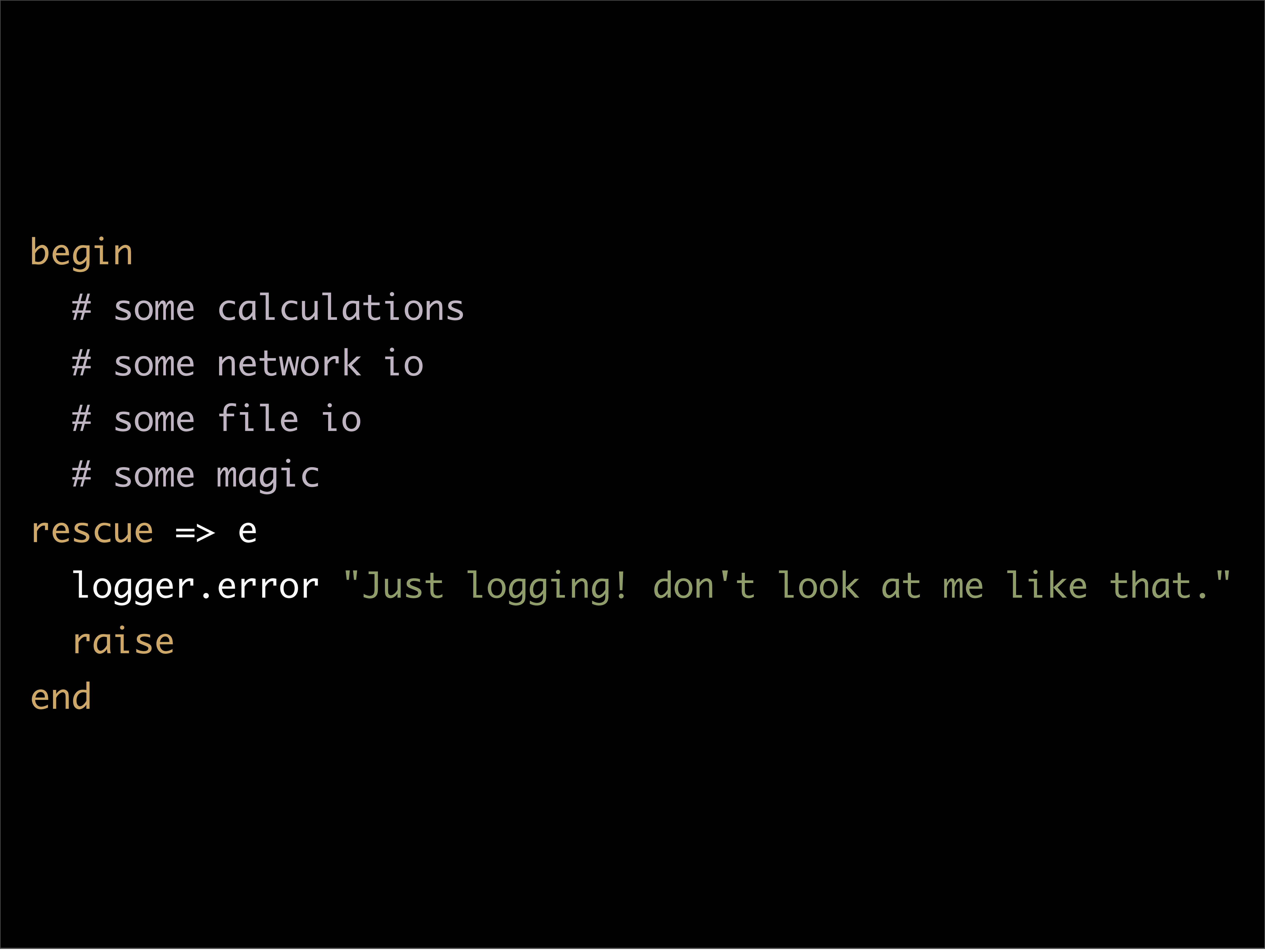


```ruby
begin
  # some calculations
  # some network io
  # some file io
  # some magic
rescue => e
  logger.error('Run! Run! we have an error!')
end
```

A classic example of wrong exception handling is unqualified rescue

If you are expecting a piece of code to raise an exception, it is always better to catch that particular exception instead of catching a generic exception.

If you are forced to write an unqualified rescue because the code block you are rescuing raises a lot of exceptions, then probably that problem should be solved first.



```ruby
begin
  # some calculations
  # some network io
  # some file io
  # some magic
rescue => e
  logger.error "Just logging! don't look at me like that."
  raise
end
```

One of the scenario where unqualified rescue makes sense is that we just want to log the exception & propagate it further.

```ruby
begin
  # some calculations
  # some network io
  # some file io
  # some magic
rescue ZeroDivisionError
  # handle the zero devision exception
rescue SomeCrazyException
  # handle the crazy exception
rescue => e
  logger.error "No idea what's heppening. I'm just going to retry in a while. #{e.message}"
end
```

Another scenario where I will have an unqualified raise is when I don’t want an unexpected exception to kill the deamon process.

I would handle all the known exceptions if I can, and in the end just suppress the exception after logging it.

Blind rescue missions

(Gregory Brown aka @seacreature)

There are just too many anti-patterns for rescue..

Another `clever` use of rescue is to assign default values in case of exceptions. What Gregory Brown calls as Blind Rescue Mission in his book Ruby Best Practices. The example in his book is so apt, that I have shamelessly copied the same thing.

```ruby
name = @user.first_name ? @user.first_name.capitalize : "Anonymous"
```

The writers intention here is to return a string “Anonymous” when user does not have first_name. This line of code is slightly complicated because of the ternary operator.

```ruby
name = @user.first_name.capitalize rescue "Anonymous"
```

Few people prefer it’s concise & questionably though but cleaner version.

certainly looks a lot more cleaner but what happens when the user itself is nil. The line still succeeds & tells me that the current user is Anonymous.

random facts

rescue => e
doesn’t rescue Exception
it only rescues StandardError

[Handoff]

*args/hash

[achamian]

Ruby supports accepting variable number of arguments or hash as a last argument of the method call. While writing a library code this comes in handy.

```ruby
def accepts_var_args(*args)
  planet = args[0]
  no_of_moons = args[1]
  moons = args[2..(no_of_moons + 1)]
  continents = [(no_of_moons + 2)..-1]
end
```

But when you are dealing with non-generic code it is better not to accept the var args. Explicitly calling out list of arguments to the method might increase the method signature but it is much more readable.

large number of parameters

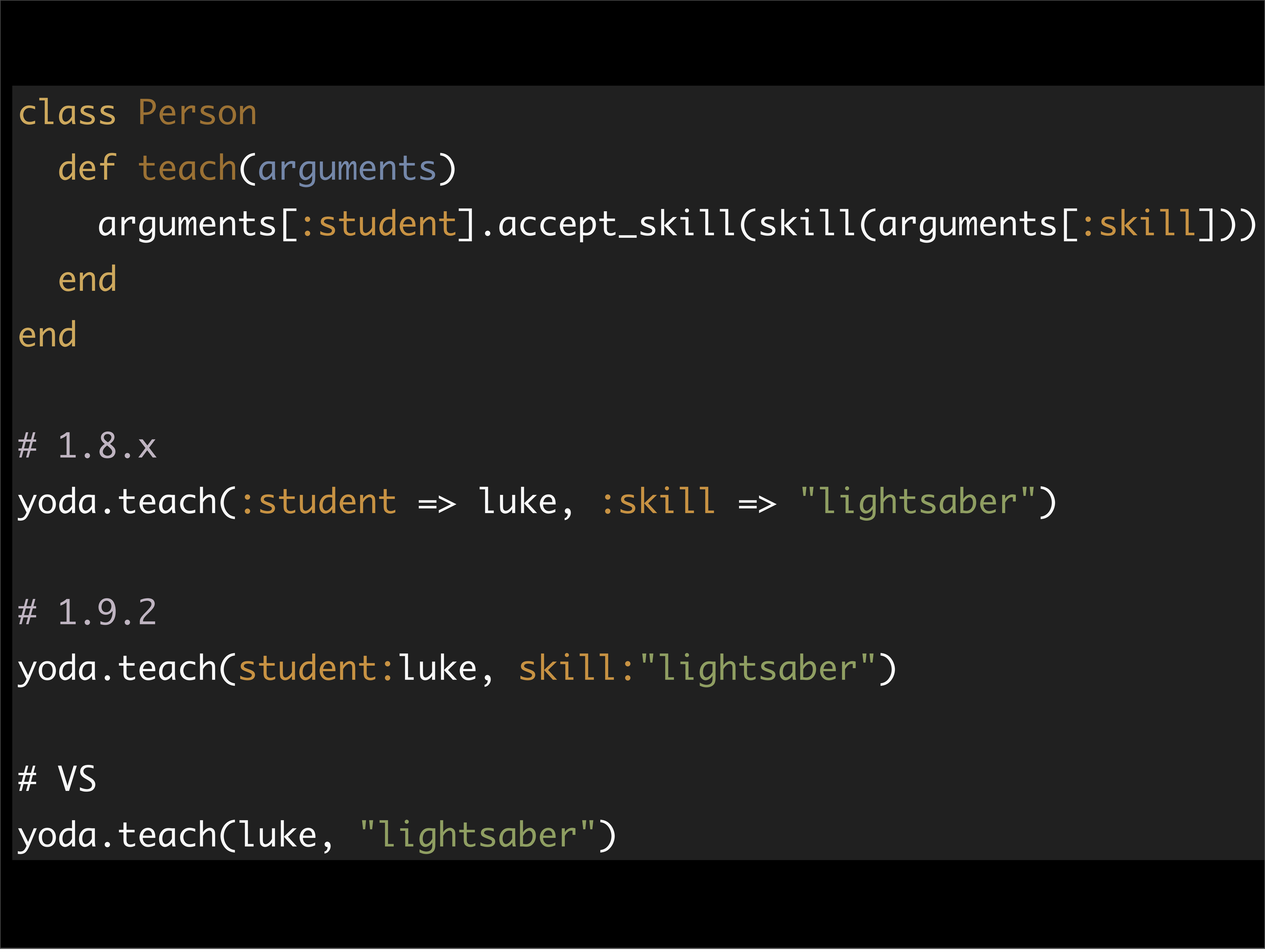
If method is accepting a huge list of parameters, say 5 or more, because of which you want to shift to switch to *args, that itself is a smell.

It might be necessary for a factory or an object builder, but other places you might want to look at that list and see if those arguments can be clubbed into one or more objects. I am not talking about DTOs here. These related parameters almost always represent some job which method under consideration is doing which can be pushed to that object itself.

hash as the last argument

Same can be said for the hash as a last argument of the method. If the hash represents a long list of options which user may or may not pass, I can understand accepting a hash as it makes it easier to invoke method.

Typical example of such options hash is any of the Rails view helper methods.



```ruby
class Person
  def teach(arguments)
    arguments[:student].accept_skill(skill(arguments[:skill]))
  end
end

# 1.8.x
yoda.teach(:student => luke, :skill => "lightsaber")

# 1.9.2
yoda.teach(student:luke, skill:"lightsaber")

# VS
yoda.teach(luke, "lightsaber")
```

Here the first approach looks tempting as the invocation is lot more readable than the second case. But I'ld rather wait for Matz to support such method invocations over using hashes.

Better state the number of arguments method accepts in the method signature rather than the rdoc while we are talking about the options hash do not

modify options hash

Hamster

or even worse

and expect invoker to fetch data out of it post call

it leads to the dark side

Path to the dark side it leads to

```
def activate_skynet(options)
  options[:instructions] = "Eliminate John Conner"
  uri = "https://sky.net/activate".to_uri
  uri.post(options[:auth_headers])
end

options = {"Authentication" => "Basic: XXXXXXXXXXX"}
activation_status = activate_skynet(options)
next_step = options[:instructions]
```

please don’t

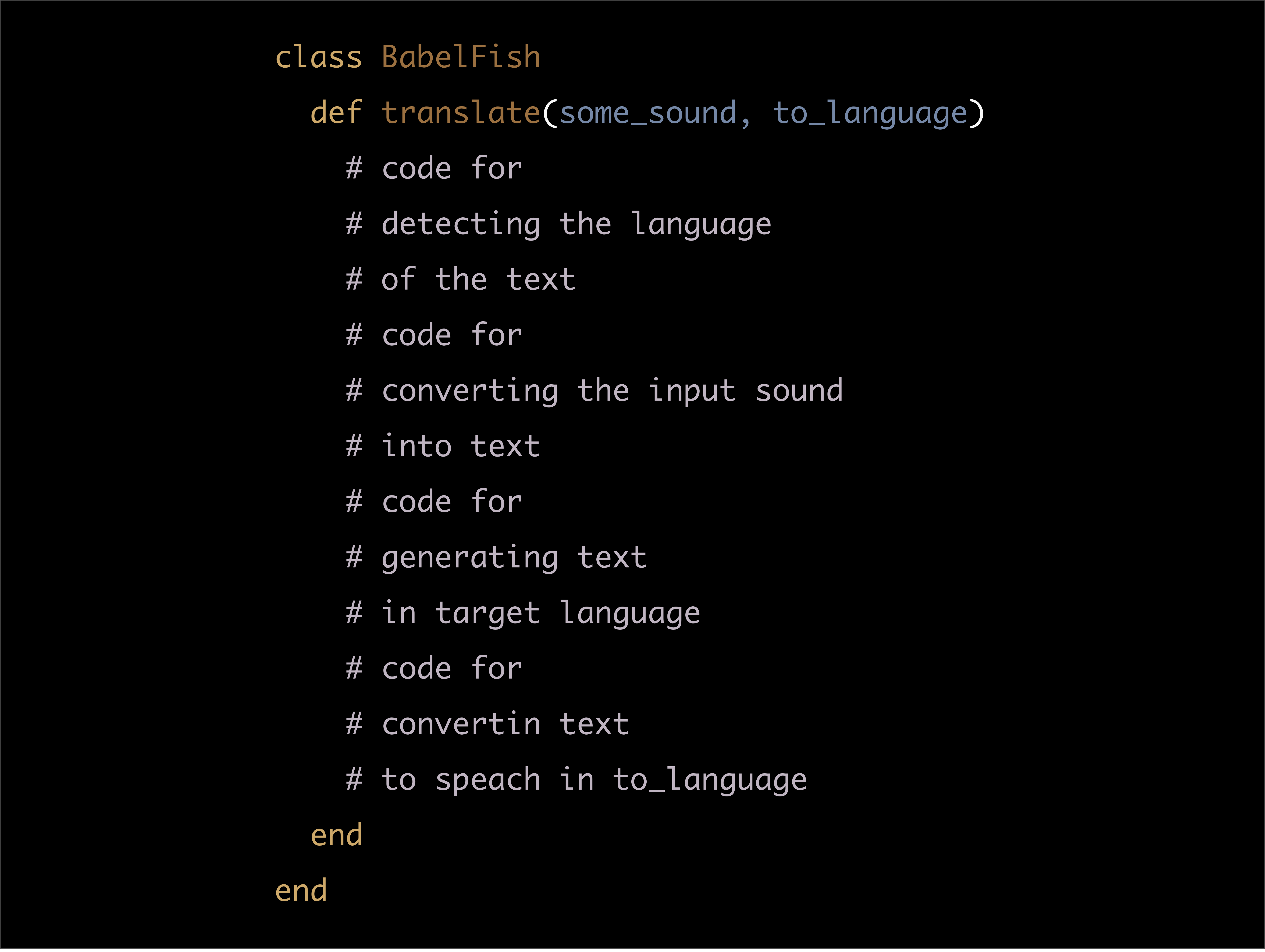
private methods

Before I put forward my opinions about private methods let's see what the proponents of private methods say about it. The top reasons why people choose private methods are:

readability

encapsulation

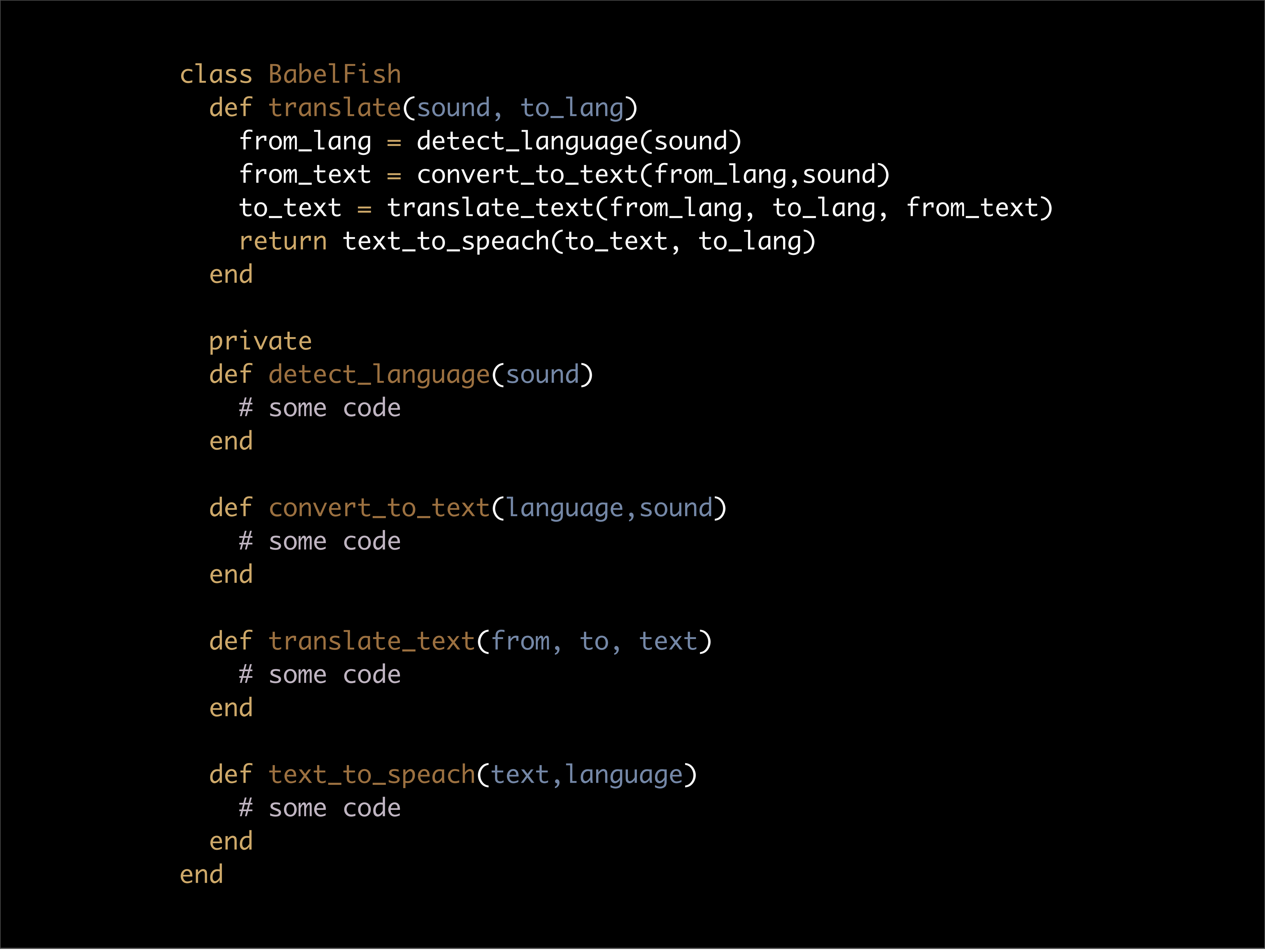
cleaner API



```
class BabelFish
  def translate(some_sound, to_language)
    # code for
    # detecting the language
    # of the text
    # code for
    # converting the input sound
    # into text
    # code for
    # generating text
    # in target language
    # code for
    # convertin text
    # to speach in to_language
  end
end
```

I have twisted the implementation of BabelFish to make it more understandable for people who are not familiar with Hitchhikers Guide. So please pardon me for this stupid code.

As I said earlier, this big method is not readable & to any new developer the implementation is not very clear. So let’s split this into multiple methods.



```ruby
class BabelFish
  def translate(sound, to_lang)
    from_lang = detect_language(sound)
    from_text = convert_to_text(from_lang,sound)
    to_text = translate_text(from_lang, to_lang, from_text)
    return text_to_speach(to_text, to_lang)
  end

  private
  def detect_language(sound)
    # some code
  end

  def convert_to_text(language,sound)
    # some code
  end

  def translate_text(from, to, text)
    # some code
  end

  def text_to_speach(text,language)
    # some code
  end
end
```

As expected the translate method has become concise & readable. The extracted methods are obviously private, as no one should be able to call detect_language or text_to_speach on BabelFish, it's just bad API & I couldn't agree more.

So now we have a clean API, with great encapsulation & all the methods are perfectly readable.

If using private methods allows me to achieve all these important qualities of a good API, why do I smell something fishy here?

then why not?

I still feel I shouldn’t have done this!

probably they don't belong here

Our BabelFish is too big & doing too many things. Any implementation change, bug fixes, new functionalities will touch this one file. Not a good thing to do.

what’s the alternative?

There may not always be an alternative to private methods, but in my experience if I look hard I have generally found another hidden entity, who can take this responsibility and make the original class smaller & neater.

I always look at private methods as first step towards better design, the key is to observe the data private methods are manipulating. If you see a data that is being manipulated purely by private methods, you have just found the entity that is waiting to be extracted.

Generally private methods can be replaced by composition or delegation.

```ruby
class BabelFish
  def translate(sound_data, to_language)
    Sound.new(sound_data).to_text.translate
(to_language).to_sound
  end
end

class Sound #:nodoc:
  # code
end

class Language #:nodoc:
  # code
end

class Script #:nodoc:
  #code
end
```

Broken encapsulation
ways to communicate not to use internals
still I prefer it

I have probably broken library level encapsulation & exposed classes with public interfaces that I don’t want my users to use. But I would any day prefer this design over a well encapsulated BabelFish.

There are ways to communicate to user that these classes should not be used. Most simple but not very effective way would be to annotate the class or module as nodoc, so that the respective class or module won’t be documented.

Class#new as factory

new is just a method

can work as a factory

few things to keep in mind

return the same type

don’t return nil

don’t raise an exception

http://blog.sidu.in/2007/12/rubys-new-as-factory.html

class methods

where is my object?

From OO design perspective I have same set of arguments against static methods that I have against private methods. In some cases static methods are worse than private methods, because they encourage procedural programming.

I will repeat myself, that if you look hard you can certainly find the objects to whom you can delegate this responsibility.

general belief
static methods are
BAD

Next few slides are my personal opinions & I am still figuring out things. They will be very controversial, and I won't be surprised if everyone of you disagrees with me.

In my more than 4 years of experience with Java, class methods or static methods have become a taboo. I have heard similar nightmarish stories of statically typed languages where bunch of static methods live in their own global context & cause limitless problems right from bad design, hard to test code, threading issues to erratic behavior of static members due to multiple custom class loaders.

Let's see what arguments I would have presented against static methods before I started working with Ruby.

can’t inherit

can’t be abstract

can’t be polymorphic

hard to test

threading issues

It suffice to say that as a rule of thumb, just stay away from static in Java world.

does the rule of thumb apply to Ruby?

But apart from the OO thingy, none of the problems stated previously hold true with Ruby.

ruby class methods are not static

The ruby class methods are unfairly compared with Java static methods, when the Ruby class methods need not be static.

Ruby class methods are way more elegant & powerful

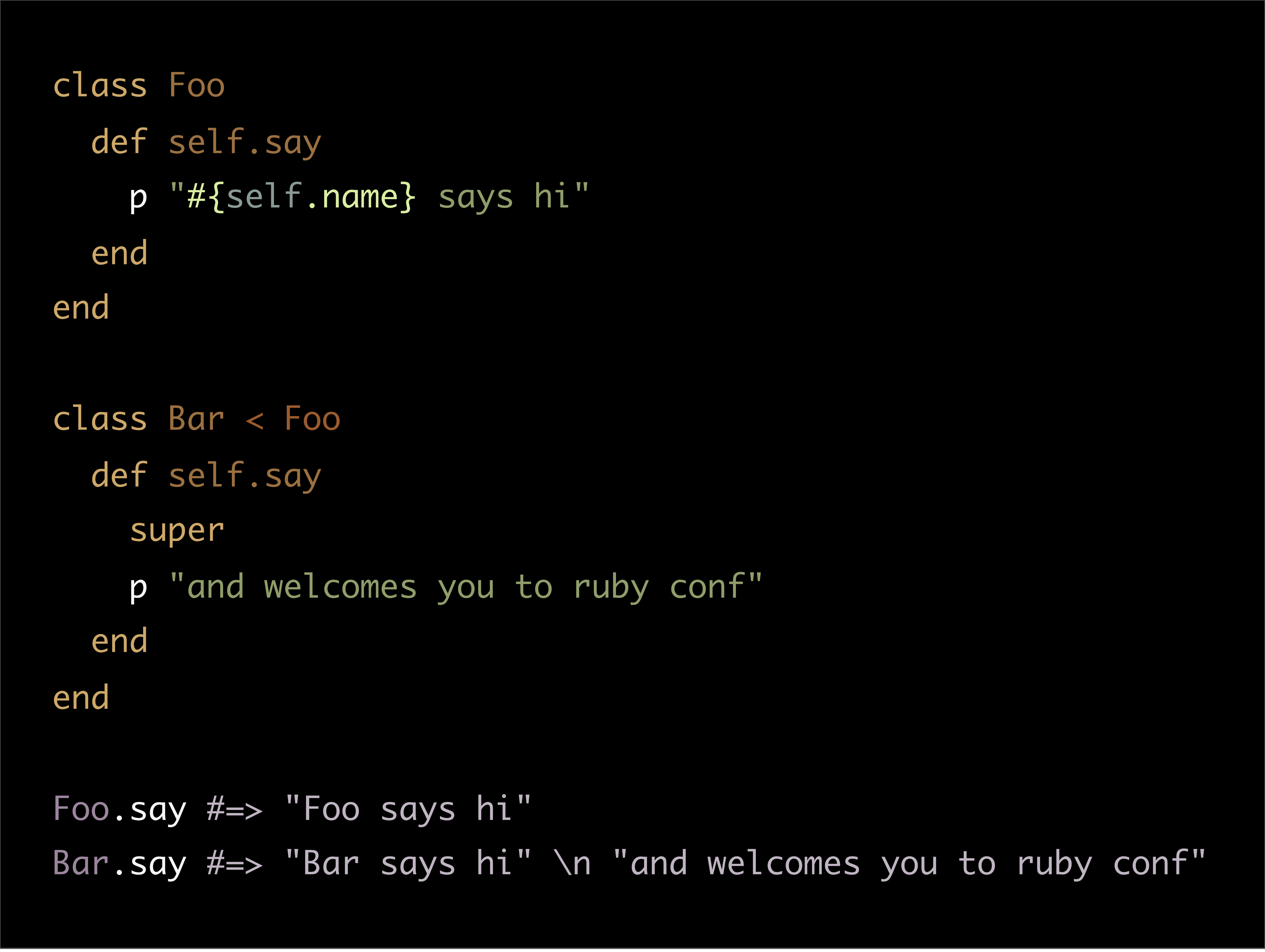
```ruby
class Foo
  def self.say
    p "#{self.name} says hi"
  end
end

class Bar < Foo
end

Foo.say #=> "Foo says hi"
Bar.say #=> "Bar says hi"
```

The ruby class methods can be inherited, making them part of the class definition hierarchy if not object hierarchy.

The method is no more a non-extensible block of static code & if coded well it can deliver a lot of value.



```ruby
class Foo
  def self.say
    p "#{self.name} says hi"
  end
end

class Bar < Foo
  def self.say
    super
    p "and welcomes you to ruby conf"
  end
end

Foo.say #=> "Foo says hi"
Bar.say #=> "Bar says hi" \n "and welcomes you to ruby conf"
```

The behavior of class methods can be extended or completely overridden, making it more reusable. If there is a logic at blueprint level but not applicable at instance level, you can effectively use class methods to DRY the code.

easy to test

The ruby class methods are as easy to test as instance methods, making the maintenance a non issue.

ActiveRecord::Base.find

ActiveRecords find method is one my most favorite example of a good class method. There will be a lot of people here who hate the kind of baggage extending activerecord pushes on simple models. But it really changed the way I used to look at class level methods, coming from Java world.

With ruby's class methods, I am more comfortable having this as class method on my models, rather than adding boilerplate code of DAO's.

methods on instance of a class Class

What I am about to say sounds slightly convoluted, but I will speak slowly. Ruby class methods are not static they are methods on an instance of a class Class def self.# not so bad

as long as it belongs

```ruby
class Company
  def sort_emplyees(all_employees)
    all_employees.sort_by(&:salary)
  end
end
```

Having a class method which does not use its state is as stupid as having an instance of Company

metaprogramming

[achamian]

random facts

alias_method
creates a deep clone of a method it’s not equivalent to
delegate

conclusion

try new things

be nice to people who'll inherit your code write tests

42

Aakash Dharmadhikari	Niranjan Paranjape
http://twitter.com/aakashd https://github.com/aakashd	http://twitter.com/niranjan_p https://github.com/achamian