THE GOOD THE BAD AND THE ABSENT

THE GOOD THE BAD AND THE ABSENT

Success Failure Nil

Shamefully Stealing From

- Sandi Metz' "Suspicions of Nil" http://www.sandimetz.com/blog/2014/12/19/suspicions-of-nil
- Avdi Grimm's "Null Objects and Falsiness" http://devblog.avdi.org/2011/05/30/null-objects-and-falsiness/

Who is hounded by this error?

Question Time

NoMethodError: undefined method `oh_god_why_this_again' for nil:NilClass

What is Nil?

Nil

```
> nil.class
=> NilClass
```

Nil

Nil is an instance of NilClass.

It's the same **nil** throughout the system. Same with **1**, **true** or **:foo**.

```
> nil.class
=> NilClass
```

```
> nil == nil
=> true

> nil.object_id == nil.object_id
=> true
```

Nil

Nil has some methods. A few strange ones (^? &?), but mostly conversion functions to coerce it to "missing" values like "" or {}.

```
> nil.instance_methods(false).sort
=> [
  ...
  :inspect,
  :nil?,
  ...
  :to_a,
  :to_c,
  :to_f,
  :to_h,
  :to_i,
  :to_r,
  :to_s,
  ...
]
```

There are problems with Nil.

Problems with Nil

Well. Hold on. Maybe there are problems with the way we **use** nil.

the way we use

Problems with Nil?

Maybe it's just us.

```
def add(a, b)
  a + b
end
```

```
add(1, 2)    # => 3
```

Say, let's have a function. This is straight out of our production app; please don't tell my boss I'm showing you this.

It adds two numbers together. 1 + 2 is 3. Great stuff.

Maybe it's just us.

```
def add(a, b)
  a + b
end
```

```
add(1, 2)   # => 3
add(1, nil) # => Kaboom
```

How about 1 and nil and... damn.

That exploded. Can't add Nil to Fixnum.

Maybe it's just us.

```
def add(a, b)
  a + b
end
```

```
add(1, 2)    # => 3
add(1, nil) # => Kaboom
add(1, "Nope") # => Kaboom
```

Same with String.

Maybe it's just us.

```
def add(a, b)
  a + b
end
```

```
add(1, 2)    # => 3
add(1, nil) # => Kaboom
add(1, "Nope") # => Kaboom
add(1, ActionDispatch::Routing::Mapper.new)
```

And whatever the hell this is.

Maybe it's just us.

```
totals =
  VisitStats.fetch_previous_week(
    Time.now # ending at...
  )

# => [1, 20, 300, 2, 7, 1, 20]
```

Okay, different example. We have a site, and it has something that gives us back a list of visits per day over the last week.

Today we got 1 visit, three days ago we got featured on TechCrunch...

Maybe it's just us.

```
totals =
  VisitStats.fetch_previous_week(
    Time.now # ending at...
  )

week = 0
totals.each do |visits|
  week += visits
end
```

I just want to add the number of visits together for this week. No big deal.

And yes, if you're grimacing at this, we can transform it into...

Maybe it's just us.

```
totals =
  VisitStats.fetch_previous_week(
    Time.now # ending at...
  )

week = totals.inject(0) {|s,visits|
  s + visits
end
```

... this. Same thing. Whatever.

So we're adding up these visits.

Maybe it's just us.

```
totals =
  [1, 20, 300, 2, 7, 1, 20]
```

```
week = totals.inject(0) {|s,visits|
  s + visits
end
```

And that works just fine. We have our totals from before, and it all adds up to...

Maybe it's just us.

```
totals =
  [1, 20, 300, 2, 7, 1, 20]
```

```
week = totals.inject(0) {|s,visits|
  s + visits
end
# => 351
```

Maybe it's just us.

```
totals =
  [1, 20, nil, 2, 7, 1, 20]
```

```
week = totals.inject(0) {|s,visits|
  s + visits
end
```

But throw a nil in there. Maybe we didn't have any visits that day. We're representing the **absence** of visits, right?

Maybe it's just us.

```
totals =
  [1, 20, nil, 2, 7, 1, 20]
```

```
week = totals.inject(0) {|s,visits|
  s + visits
end
# => TypeError: nil can't be coerced into Fixnum
```

Maybe it's just us.

```
totals =
  [1, 20, nil, 2, 7, 12, 20]

week = totals.inject(0) {|s,visits|
  s + visits
end
# => TypeError: nil can't be
```

ALL YOUR FAULT.

YOU'RE JUST A BAD PERSON.

You did the code wrong!

You should have been coding defensively!

Bad developer, no biscuit.

Maybe it's just us.

```
totals =
  [1, 20, nil, 2, 7, 1, 20]

week = totals.inject(0) {|sum,visits|
  if visits
    sum + visits
  else
    sum
  end
end
```

You should have done this instead. Now the code will work!

Maybe it's just us.

```
totals =
  [1, 20, "300", 2, 7, 1, 20]

week = totals.inject(0) {|sum,visits|
  sum + visits.to_i
end
```

Well, okay. But what about if we get a string? We can be defensive about that too!

Maybe it's just us.

```
totals =
  [1, 20, "300", 2, 7, 1, 20]

week = totals.inject(0) {|sum,visits|
  sum + visits.to_i
end
```

Stop a minute. This is about as ridiculous as getting back something like a, say, ...

Maybe it's just us.

```
totals =
  [1, 20, FizzBuzz.new, 2, 7, 1, 20]

begin
  week = totals.inject(0) {|sum,visits|
    num = Integer(visits)
    sum + visits.to_i
  end
rescue ArgumentError =>
  ...
end
```

... FizzBuzz in the middle of your list. It doesn't make sense.

And yes, we can defend against /that/ by doing assert()-like things...

Maybe it's just us.

```
totals =
  [1, 20, FizzBuzz.new, 2, 7, 1, 20]

begin
  week = totals.inject(0) {|sum,visits|
    num = Integer(visits)
    sum + visits.to_i
  end
rescue ArgumentError =>
  ...
end
```

... and catching the exception here or whatever, but that **sucks**.

It's This.

```
totals =
  VisitStats.fetch_previous_week(
    Time.now # ending at...
  )
```

This is the problem.

This function right here.

In our [1, 20, nil, ...] example, it's returning to different **kinds** of things.

It's This.

```
totals =
  VisitStats.fetch_previous_week(
    Time.now # ending at...
  )

# Time -> [FixNum]
```

If we had a way of annotating what we were expecting out of this function, it might look like this.

It's This.

```
totals =
  VisitStats.fetch_previous_week(
    Time.now # ending at...
  )

# Time -> [FixNum]
```

It takes a Time. It's a class. It's a kind of thing. It's a **type**.

It's This.

```
totals =
  VisitStats.fetch_previous_week(
    Time.now # ending at...
  )

# Time -> [FixNum]
```

... And it returns another type of thing.

It returns a **List** of **FixNums** (numbers).

This promise is wrong, because it's sometimes handing us Nils.

Kind of a Big Deal.

Nils, Strings, Time, or any other class are **types**.

When you have these thing that hand back Nil, or String, suddenly anything calling those functions in your code suddenly sprouts an **if** or a **.to_???** cast, and has to account for the different types it's getting back.

Kind of a Big Deal.

All of this added complexity from checking compounds.

It's arguably ***one*** of the things that makes it difficult to manage Ruby codebases as they grow in size; they hand out nil or other unanticipated types, and everything festers as a result.

Kind of a Big Deal.

Instead of handing back Nils sometimes, without telling anyone, you need to make a deliberate choice as to where to use them.

Once you've fallen into the hole, it's very difficult to get back out with the tools that we have available to us when we work with Ruby.

Hacker News Onion
@HackerNewsOnion

Follow

Developer who inherited 5-year-old Rails codebase secretly hoping for company collapse

RETWEETS 452

FAVORITES 335

7:18 AM - 12 Jun 2014

Actual Problems with Nil

(An aside.)

Back to the earlier slide. Nil does actually have its own problems.

It's good that it's a standardised representation of Nothing that everything is familiar with, but it has special cases that make it difficult to change after you start using it. It's **too** special.

Nil/False

```
thing = ...

if thing
   "truthy"
else
   "falsey"
end
```

Here's a thing. It has a value.

If *thing* is "truthy", then we get the string "truthy" back. If not, then "falsey".

1 is truthy. 0 is truthy. "foo" is truthy. "" is truthy.

If false is falsey. If nil is falsey. And that's it.

Nil/False

```
thing = ...

if thing
   "truthy"
else
   "falsey"
end
```

You are not allowed to touch this mechanism.

There's no way to emulate "falseyness".

And this applies to **every** condition; if, while, ternary, until, …

Nil/False

```
url = site.url
# => "https://ourshop.com", or nil

if url
  "#{site.name} (at #{url})"
else
  "#{site.name} (site coming soon)"
end
```

Say we have a Site object, which has a URL that be set to "...ourshop.com", or nil.

You've probably seen this a lot. There's a thing, it's in a database table somewhere, but it doesn't have an attribute set yet.

Nil/False

```
url = site.url
# => #<URL:...>

if url
  "#{site.name} (at #{url})"
else
  "#{site.name} (site coming soon)"
end
```

And say we later want to change how we do URLs. It's something with a bit of complexity; maybe it's using something else to determine https or not. We decided to not represent it with just a String anymore.

Nil/False

```
url = site.url
# => #<URL:...>

if url
  "#{site.name} (at #{url})"
else
  "#{site.name} (site coming soon)"
end
```

With our new URL object and this *if* block, the only path we can reach is the "truthy" one.

URL can't ever pretend to be falsey.

Nil/False

```
url = site.url
# => #<URL:...>

if url.nil?
  "#{site.name} (site coming soon)"
else
  "#{site.name} (at #{url})"
end
```

The best thing we can do would be to run around and change every *if* or *while* or other condition that ever used the value to ask the object whether it's nil or falsey, so we could at least emulate it.

Nil/False

```
url = site.url
# => #<URL:...>

if url.nil?
  "#{site.name} (site coming soon)"
else
  "#{site.name} (at #{url})"
end
```

… Or write a tool to try to find them for us. Both of which are *very* difficult to do without missing cases. If we're *lucky*, our tests can assist.

Either way, it's not going to be pretty. Once you've started checking falseyness directly everywhere, changing something later gets much harder.

How can we fix all this?

(Even partially.)

It's arguable that we can't fix Nil without drastically changing the language. And we can't fix special-case falseyness.

But we have some things we can try to represent absence, or failure, or other things we're using Nil for, that are *slightly* more manageable.

Null Objects

Or, the "Null Object Pattern".

Null Objects

```
class SMTPMailer
  # ...
  def send_mail(...)
    # ... <things> ...
    true
  end
end
```

Say we have a Mailer. Its interface (as far as an eventual consumer cares) is a method called **send_mail** with a set of arguments that we don't really care about for this example.

Null Objects

```
class SMTPMailer
  # ...
  def send_mail(...)
    # ... <things> ...
    true
  end
end

class NullMailer
  # ...
  def send_mail(...)
    true
  end
end
```

We can also have a NullMailer. It matches the interface (same method name, same arguments), but does nothing except report success.

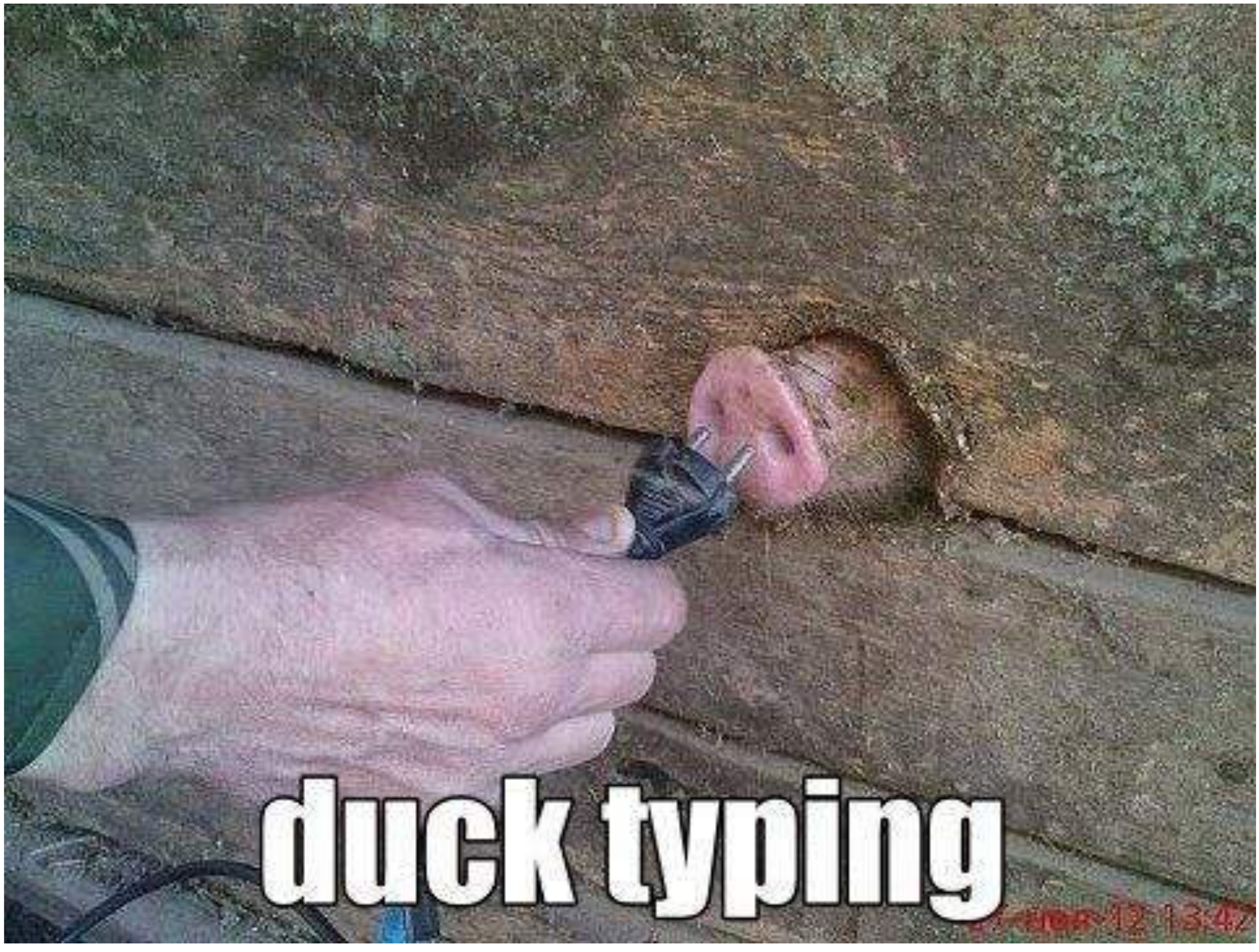
Null Objects

```
mailer = application.mailer
# => NullMailer
mailer.send_mail(...)
```

A consumer asks the application to give it a mailer.

It gets one. It uses it.

The caller is **oblivious**. Ideally, it has no way even to check if the mailer is a "null" or dummy one at all.


duck typing

Null Objects

- Has to match the interface.

It has to match the interface, and that interface can be huge. And get out of sync with everything else that implements it. You could try to use inheritance and suffer for it as things change, or write tests, or... Reimplement interface checking, basically.

(The Go people in the audience are sniggering right now.)

Null Objects

- Has to match the interface.
- "False positive" results.

Though it might be useful to get an object back you can call the same message on, it's important to **not hide failures** by returning null objects.

Nothing will realise everything is broken until it's far too late.

Null Objects

- Has to match the interface.
- "False positive" results.
- For oblivious callers.

Null Objects

```
class User < ActiveRecord::Base
  # ...
  def guest?
    false
  end
end

class GuestUser
  # ...
  def guest?
    true
  end
end
```

Here's an anti-pattern you see recommended a lot: a "logged in user" or a null "guest user". On the surface it seems fine, but now you have to:

a) Match the entire (huge) interface, and

b) Keep checking if the guest is a Real User before you can do anything meaningful.

Null Objects

```
def user_from_session(session)
  id = session[:user_id]
  if id.nil?
    GuestUser.new
  else
    FetchUser.by_id(id)
  end
end
```

Here's what using that split may be like. Have a method to get something back and...

Null Objects

```
user =
  user_from_session(session)

if user.guest? ...

...

if user.guest? ...

...

if user.guest? ...
```

... be forever checking if you got back a working user.

(You don't want to send email to a dummy user, do you?)

Domain-Specific "None" Classes

I just made this term up because I didn't know what to call it. Sorry. I'm talking about classes that are defined inside or close to whatever is performing the fallible operation.

DSNC (← I made that up.)

```
class FetchUser
  Missing = Struct.new(:id, :message)

  def self.by_id(id)
    object = ...
    ...
    if object.nil?
      Missing.new(id, reason)
    else
      object
    end
  end
end
```

Here we're returning an instance of an explicit FetchUser::Missing class (with some additional information) instead of a plain nil.

DSNC (← I made that up.)

```
class FetchUser
  Missing = Struct.new(:id, :message)

  def self.by_id(id)
    object = ...
    ...
    if object.nil?
      Missing.new(id, reason)
    else
      object
    end
  end
end
```

Anything consuming this can check the class/type, or we could add a method on Missing and check that... But it's at least swappable down the road, unlike plain nil.

Wrapping "Just(x)" or "Nothing" Classes

Or instead we can wrap things into a **container** of sorts.

We can start generalising how we want to handle operations on the container itself, instead of the contents directly.

Maybe (Just or Nothing)

```
class Maybe; end

class Just < Maybe
  attr_reader :value
  def initialize(value)
    @value = value
  end
end

class Nothing < Maybe
  def value
    self
  end
end
```

Or instead we can wrap things into a **container** of sorts.

We can start generalising how we want to handle operations on the container itself, instead of the contents directly.

Maybe (Just or Nothing)

```
class FetchUser
  def self.by_id(id)
    object = ...
    ...
    if object.nil?
      Nothing.new
    else
      Just.new(object)
    end
  end
end

user = FetchUser.by_id(...)
# => Nothing, or Just(User)
```

Maybe (Just or Nothing)

```
Maybe.new(FetchUser.by_id(...)) >-> user {
  application.mailer.send_mail(
    user.email,
    "Welcome, #{user.name}"
  )
}
# Nothing(), or a Just(sent-email-result).
```

Maybe we add operations that let us optionally proceed only if something is present.

Either (Left or Right)

```
Maybe(FetchUser.by_id(...)) >-> user {
  if user.activated?
    Left("#{user} already activated")
  else
    Right(user)
  end.fmap {|user| ActivateUser.perform(user) }
      .fmap {|user|
        application.mailer.send_mail(
          user.email,
          "Welcome, #{user.name}"
        )
      }
}
# Left(message), or
# Right(newly-activated-user)
```

And I wrote a big slide up of how you can stick these together, and then realised I had a big bug in the middle of it.

I'm keeping this in here as an example of how you could have these generic containers…

Either (Left or Right)

```
Maybe(FetchUser.by_id(...)) >-> user {
  if user.activated?
    Left("#{user} already activated")
  else
    Right(user)
  end.fmap {|user| ActivateUser.perform(user) }
     .fmap {|user|
       application.mailer.send_mail(
         user.email,
         "Welcome, #{user.name}"
       )
     }
}
# Left(message), or
# Right(newly-activated-user)
```

But I wouldn't advocate their use with Ruby.

Have a look at Swift, Rust, Scala, Haskell for the painless version of all this.

If you really want to.

Kleisli

http://blog.txus.io/kleisli/

This is not an endorsement.

If you really want to.

Kleisli

http://blog.txus.io/kleisli/

Neither is this.

The Elephant.

To actually **solve** this, we need a way to enforce not-nilness, and a way to check our program in advance.

We have neither. These are all band-aids.

Summing up.

Summing up.

- There are types of things. Nil is another type, but we sometimes forget that.
- Only Nil and False can be falsey. Special rules apply, which sucks.
- Null Objects for when the caller is oblivious.
- Try to model missing or bad results, rather than throwing back nil and nothing else.

Fin.

Rob Howard
@damncabbage
robhoward.id.au