



Hello! My name is John Crepezzi - I've been doing Rails for a long time at a bunch of different companies. Like last year, I'm not going to spend much time up here talking about myself, because I'm so excited about my talk and I ran out of room. You'll just have to trust me that I'm pretty cool.

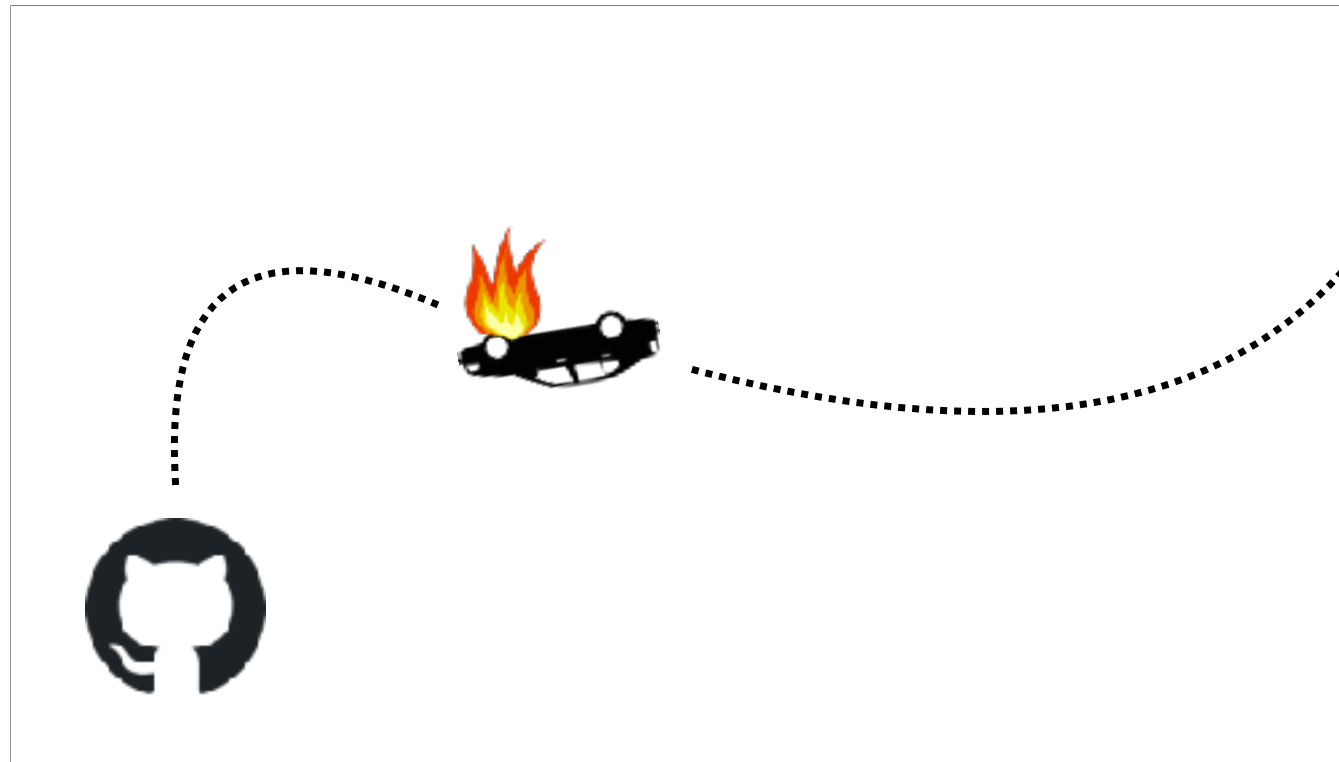
SEEJOHNRUN



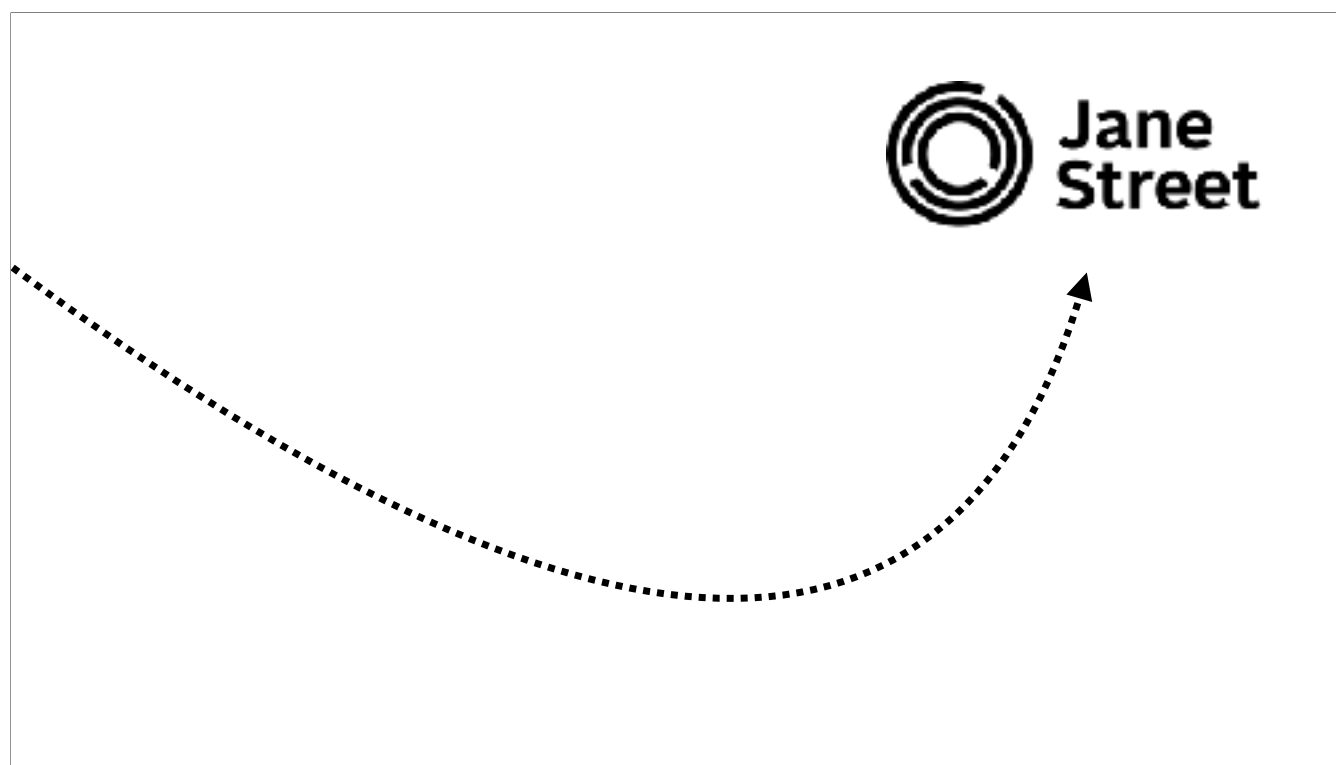
If you're interested in learning more about me please consider following me. I'm @seejohnrun on GitHub and Twitter. Hell I'm on Mastadon but I only have 3 followers, so find me over on mastadon social even though I try to make a habit of not tooting in public



Like a lot of you I hope, I love Ruby and I love Rails. I love it even more than last year! Who here loves Rails more than last year? It's a web framework that lets me easily and safely express my thoughts with no unnecessary fluff.. on top of a programming language I find to be extremely beautiful.



Last year around the time of RailsConf in Portland, I left my job at GitHub.. after a brief misstep somewhere else



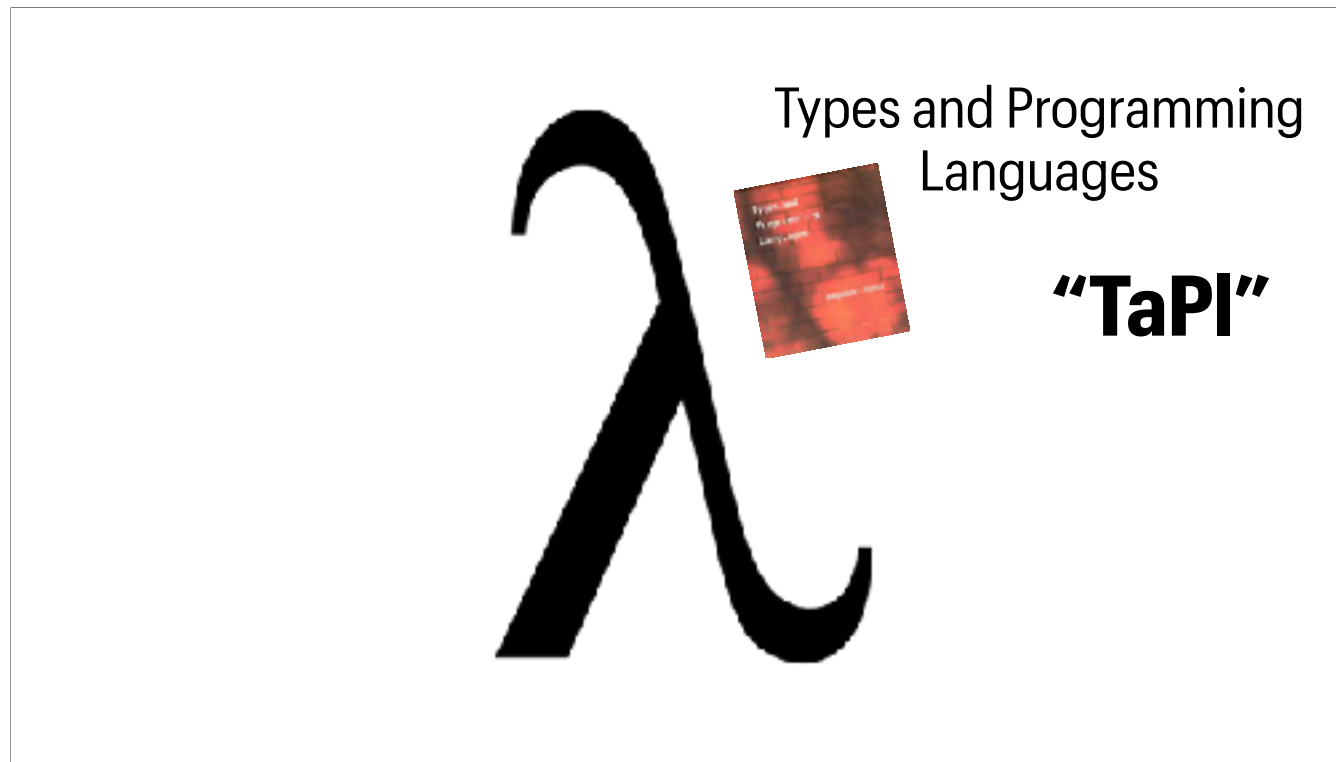
I now work at Jane Street, a tech-forward finance company based in New York. At Jane Street I continue to work on developer tooling and it's pretty exciting because they have some of the best developer tooling I've ever seen. Robust systems for code review, deep editor integrations, fast build pipelines, and some amazingly intelligent people who really know how the software they write works at a base level I've not seen before. What they don't have.. is Ruby and Rails.



Jane Street writes nearly all of their code in a fairly obscure programming language called OCaml. I've never worked in OCaml, not many other companies or projects use it, and to be honest I stressed quite a bit before joining that I might be putting myself in a situation where I'd spend years investing my time in a skill that would largely not be transferable.



When you first join the company you go through a 2-week long onboarding to the language. It's not something you'd want to miss because.. well because it's fun and also required but. But also because you'll absolutely need it. OCaml has features and patterns that don't make a ton of sense if you haven't been exposed to them before.



Someone’s going to walk over to you and reference the lambda calculus. You’ll make a confused look, and they’ll probably hand you this book: (CLICK) Types and Programming Languages which they’ll refer to as TaPI and say something like “It’s a classic”.

This is clearly a company that has thought a lot about their choice of programming language.

REMINDERS TO SELF

- **Keep an open mind**
- **Be ready to learn**
- **Strengthen mental toolkit**

The impressions left on me were:

I need to keep an open mind and not just talk about Ruby all day

I need to figure out what they know that I don't know

Given that Ruby and OCaml exist in such separate worlds, what new tools can I gain that might apply in other places?

OCAML != RUBY

In this talk I'm going to go over some neat things from OCaml and show how they can be applied to Ruby. What I'm NOT going to try to do, the thing so many talks about other technologies at these conferences do, is convince you that OCaml is in some way better than Ruby. Spoiler alert: I don't think it is!



I convinced my wife to name one my kids Ruby, I don't have any named OCaml. Rather, I hope to demonstrate that a lot of the things we see as features of OCaml are actually just design patterns and with a little work we can use them in Ruby too. The literal wall in my house is colored Ruby.

(this slide intentionally left blank)

Okay so let's get started, and to set a good basis, we should probably go over some quick similarities and differences between these two languages.



DYNAMIC



STATIC

The first and most obvious difference is the style of the type system. Ruby's type system is dynamic and OCaml's is static. For those of you that haven't programmed in a statically typed language, this means that the type of each variable is known before the code runs.

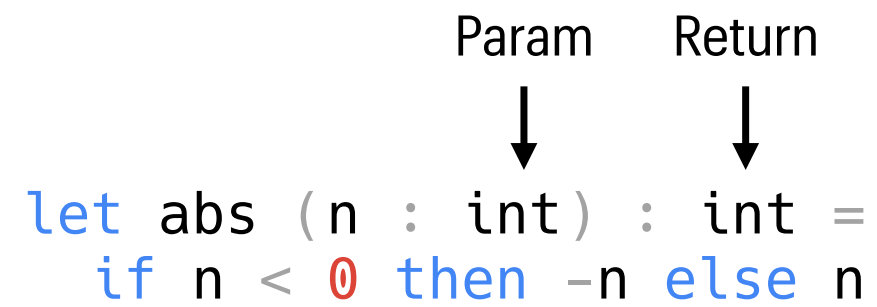
```
var = 5  
  
var = var.to_s # "5"  
  
var = var.to_sym # :5  
  
var = params[:var] # ???
```

Consider this example. First we define a variable called `var` to hold the integer 5. Then we redefine `var` to be 5 but as a string, and then the same but as a symbol. Finally we take a value for `var` from a param, and now as we read this we don't even know what type ``var`` could be. This is perfectly valid syntax and will run. We've all written code like this, it doesn't even feel weird.. but, what is the type of ``var``? The answer is it kinda depends on where you are in the execution! We can't look at the code and assign `var` a single type. The type of the variable is only determinable so to speak, at "run time".

```
"hello world".abs
```

```
Traceback (most recent call last):  
  undefined method `abs' for "hello world":String (NoMethodError)
```

It's even pretty easy for me to write some code that doesn't make sense in Ruby. Here I'm calling `abs` to try to get the absolute value of a `String`. This code, we can all look at it and know that it's wrong - in most codebases String doesn't have a method called abs, but that doesn't stop Ruby from running it!



```
let abs (n : int) : int =  
  if n < 0 then -n else n
```

By contrast, in a statically typed language like OCaml, types are always known. We tell the compiler the types of things by adding type annotations - code that declares the type of variables. You also go and declare for each function you define what types they take in as parameters, and what type they return.


```
let var = ref 5 in  
var := "5"
```

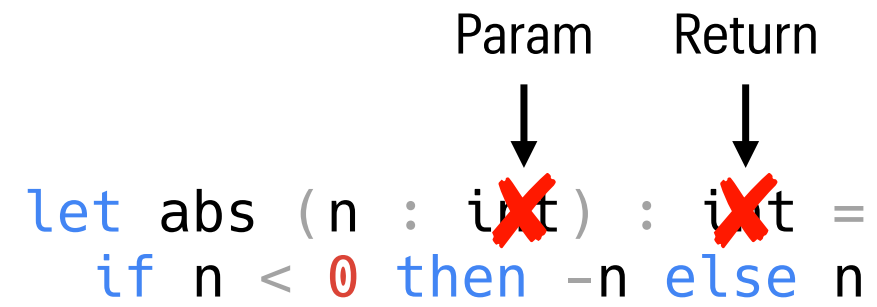
```
File "sum.ml", line 6, characters 9-12:  
6 |   var := "5";  
      ^^^  
Error: This expression has type string but an expression was expected of type  
      int
```

You can't change the type of a variable in OCaml. This isn't something where if you do it you get an error - instead, the entire program will refuse to even compile and thus you won't be able to run it at all.

abs "42"

```
File "sum.ml", line 7, characters 6-10:  
7 |   abs "42"  
      ^^^^  
Error: This expression has type string but an expression was expected of type  
      int
```

Extending that a bit more, you actually can't even write code that mixes up types. Here's I'm trying to pass our abs function a string instead of an int and the OCaml compiler is refusing to do its job because it knows that abs only takes a single int parameter.



The diagram shows the OCamL code for an absolute value function. Above the code, the words "Param" and "Return" are written. Arrows point from "Param" to the parameter type `int` in the function signature, and from "Return" to the return type `int`. Both `int` types are crossed out with a large red 'X'.

```
let abs (n : int) : int =  
  if n < 0 then -n else n
```

OCaml has a powerful type inference engine, and that means that you often don't have to write these types. The OCaml compiler, though, is still doing its work. Even if you don't write them, every variable and function declaration in your code (if it compiles) is known before the program starts.

(this slide intentionally left blank)

Another big difference is in their programming style or paradigm. To people fairly new to programming, this may seem like a pretty weird thing.



Like, I program 'fakie style'

```
# I like all types of 🍕  
def print_pizza_truths(*toppings)  
  # Go through each topping 🧀 🍅 🌶️  
  toppings.each do |topping|  
    # 👷 Build a string  
    statement = "I love #{topping} on my pizza!"  
    # Tell the 🌍  
    puts statement  
  end  
end
```

or “my style involves lots of comments that include emojis”.

But these are real things, and each as it was made was a revolution in how code was written and reasoned about.

PARADIGMS

- **Imperative**
- **Procedural**
- **Object-Oriented**
- **Functional**

Generally the most common paradigms worth discussing are imperative, procedural, object-oriented, functional. Of course there are a few more, everyone will debate you about the lines between them, and technically some of these are precursors to each other but these are the main ones when we're talking about programming languages that you might use for an everyday project.

```
puts "hello railsconf"  
puts "hello railsconf"  
puts "hello railsconf"  
puts "hello railsconf"  
puts "hello railsconf"  
puts "hello railsconf"  
puts "hello railsconf"
```

Imperative languages like Assembly and the original BASIC are execute statements in order, there are no functions and any sharing of logic typically relies on jumping meaning using something like goto.


```
require 'goto'

frame_start

i = 0
label :loop do
  i += 1
  puts "hello railsconf"
  goto :loop if i < 7
end

frame_end
```

We can actually use something like goto in Ruby using this library from Benjamin Bock. Notice we're not making use of scope to hide variables, and we're creating a loop by jumping between parts of our code

```
def print_hello  
  puts "hello railsconf"  
end  
  
for i in 1..7 do  
  print_hello  
end
```

Procedural languages like C organize imperative code into subroutines, or functions. This allows us to better organize code, isolate local variables to the functions they're created in and avoid mistakes like jumping outside of our program.

```
class Printer
  def initialize(count)
    @count = count
  end

  def run
    for i in 1..7 do
      puts "hello railsconf"
    end
  end
end

Printer.new(5).run
```

Object-oriented languages like Ruby or Java organize behavior into objects, and then allows sending messages to those objects. So instead of just having functions that take parameters, there's kinda this implicit parameter, call "self" on each function, which we now have to refer to as 'methods'.

```
# Recursive
def print_messages(count)
  return if count == 0
  puts "hello railsconf"
  print_messages(count - 1)
end
print_messages 7

# Or, anonymous "functions"
7.times { puts "hello railsconf" }
```

functional languages like Haskell use functions as their primary organizational construct. Functions can be combined, passed to each other, and contain nested scope.

But wait John - you said there are styles, you told me Ruby is Object-Oriented but all of these code examples are in Ruby. That's true - and it's because Ruby is actually multi-paradigm. It supports all of these programming styles, it's just particularly good at being an object-oriented language.

In fact, most languages are these days. It turns out when new paradigms come out, they put pressure on language authors to adapt to stay relevant.



The image shows the OCaml logo, which consists of the word "OCAML" in a bold, sans-serif font. An arrow points from the word "Objective" below it to the "O" in "OCAML".

OCaml it turns out is also multi-paradigm and has first-class support for nearly all of these styles. Actually, the “O” in OCaml means Objective, which is a reference to its object-oriented features. It also has a specialty though, and OCaml is decidedly focused on being a functional language.



DYNAMIC

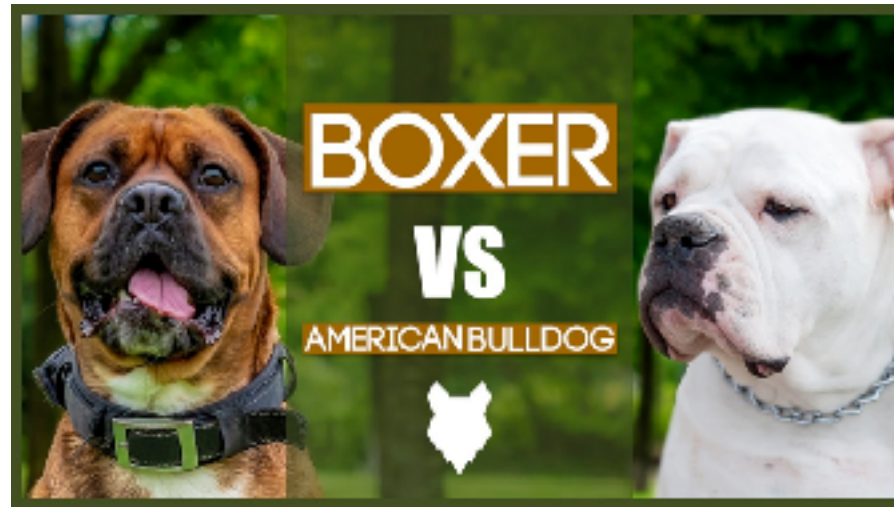
OO



STATIC

FUNCTIONAL

In the same way rubyists don't write recursive code too often, OCamlers?? tend to kinda just ignore the Object-Oriented features.



Aside from the code just looking different, the difference in style affects a programmer's default stance towards a problem they are working on. Faced with something complicated in Ruby, we'll often simplify the problem by using classes and objects to maintain state. Functional languages like OCaml will instead solve the same problem by introducing functions and the compiler is purpose built to make this type of code performant.



DYNAMIC

OO

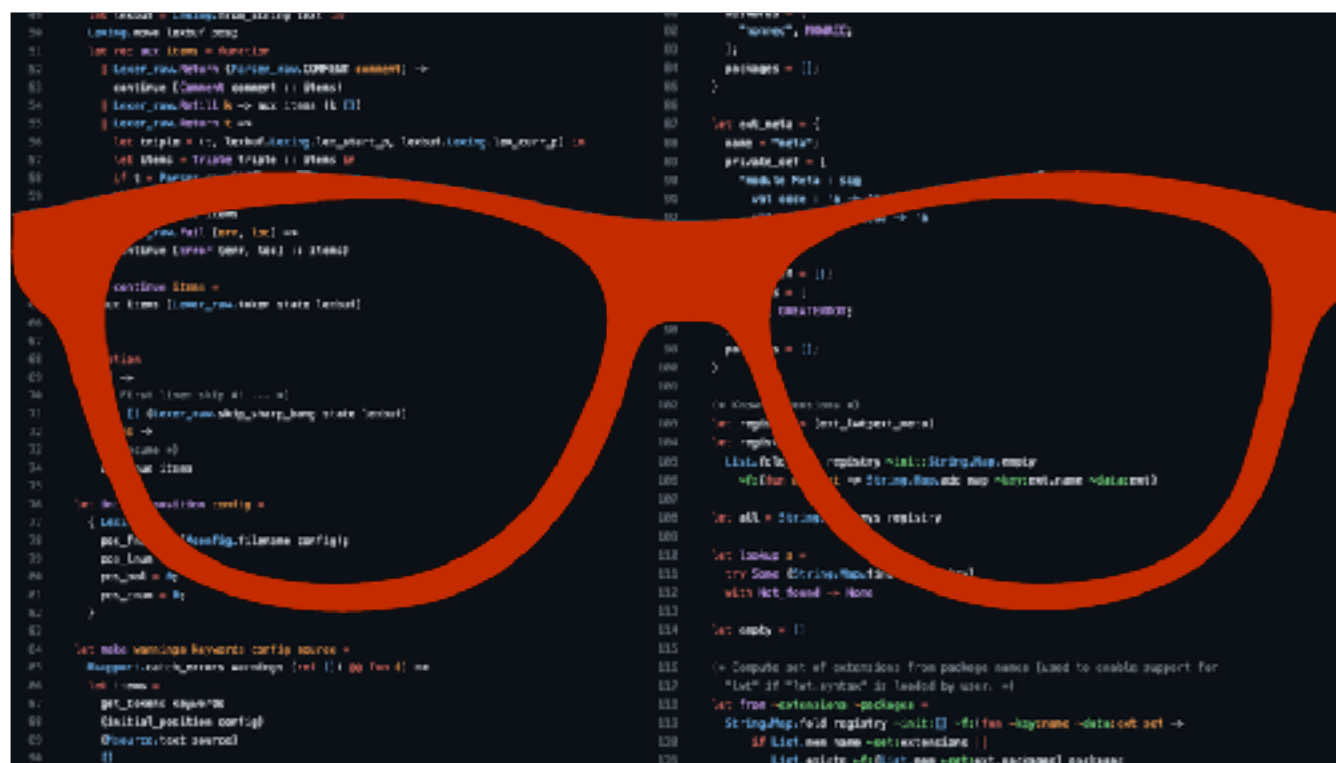


STATIC

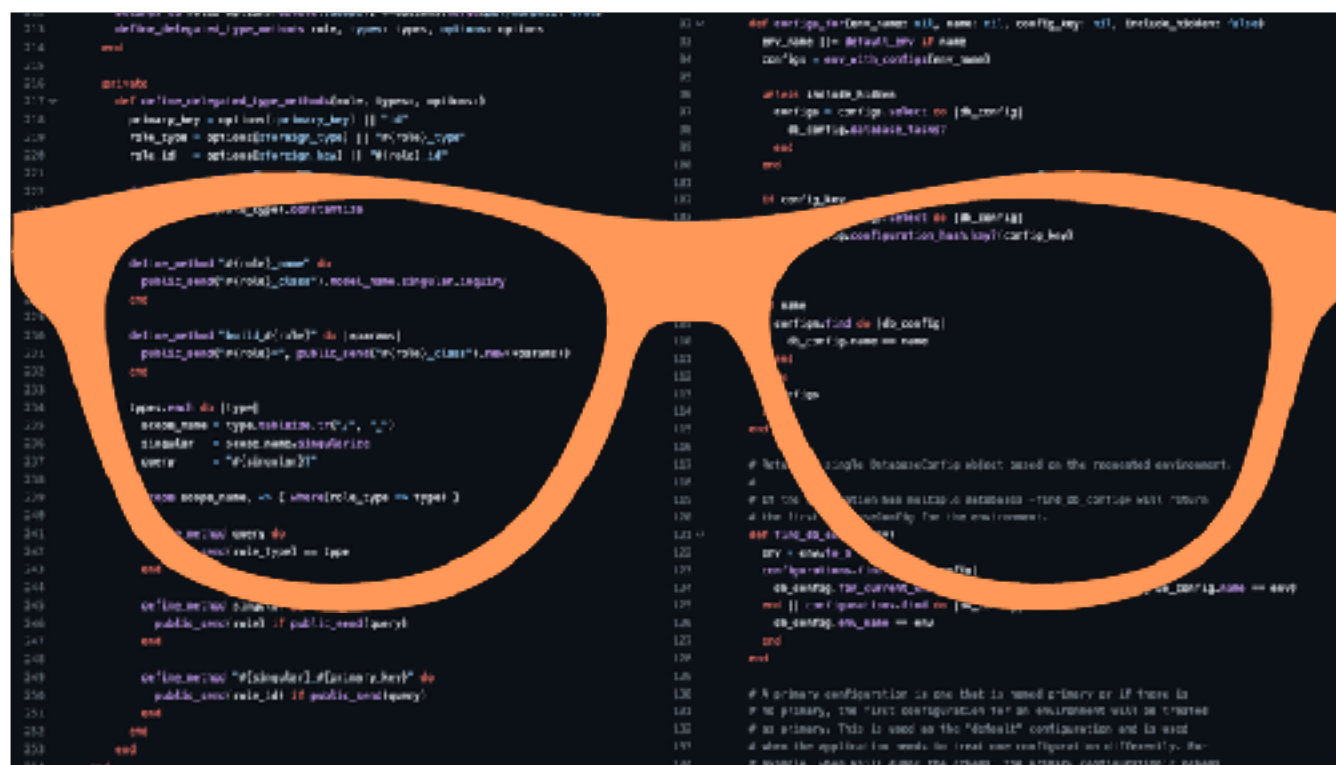
FUNCTIONAL



Striking as these differences may be, the two languages have something very important in common: they are both expressive and focused foremost on developer happiness. They just go about it differently. Ruby makes developers happy by giving them flexibility to express things the way they want. Consider how many different ways we just wrote the same code, and I'm sure any of us could come up with 10 more easily. OCaml makes developers happy by making certain types of easy-to-make mistakes no longer possible, reducing the problem space developers have to consider when writing or modifying code.



Having worked with Ruby for so long, I tend to write OCaml code with Ruby-tinted glasses on. I try to create module interfaces that read like sentences.. I am obsessive about separation of concerns for different layers of my app.. and I spend a lot of time making sure the code I'm writing is testable.



And when I get home and go back to writing Ruby, I've been noticing the same longing in the other direction. I'm missing some of the features that make me happy when writing OCaml, that put my mind at ease about certain types of problems or make refactoring easier.

3X

- **How it works in OCaml**
- **How we can use it in Ruby**
- **How it can help Rails**

So I think about how the lessons I've been learning at work can translate back into Ruby and Rails. Today I'll be covering 3 different features and for each I'll go over:

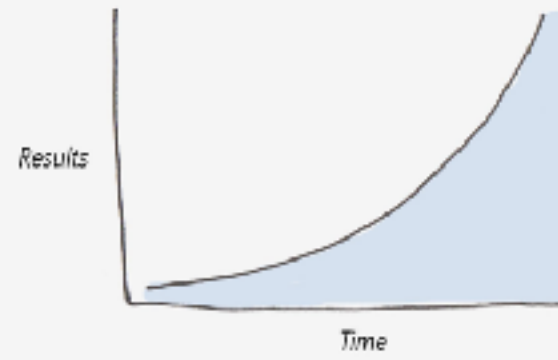
What the feature is and how it works in OCaml

How something similar can be applied in Ruby

Because we're at RailsConf: how you might use it in your Rails application

EXPONENTIAL GROWTH

Improvements come slowly in the beginning, but your gains increase rapidly over time.



JamesClear.com

I should also issue a slight word of warning. This talk started out pretty light but does ramp up fairly quickly and the pace is pretty fast. I like really technical talks, but just want to acknowledge that this is a ton of material. If there's something in here that doesn't quite make sense, consider coming back to the idea or playing around with the concept yourself. So many of these things didn't click for me the first time around.

With that out of the way, let's get started with our first topic:



probably OCaml's most fun-sounding feature: Functors.

```
(* string *)  
let s = "string" in  
  
(* int *)  
let i = 42 in  
  
(* int -> int -> int *)  
let add x y = x + y in  
  
()
```

In OCaml there are basic value types. Think like a string, integer, map, or array. and then there are functions. Like I mentioned earlier, functions take in some number of parameters and return something

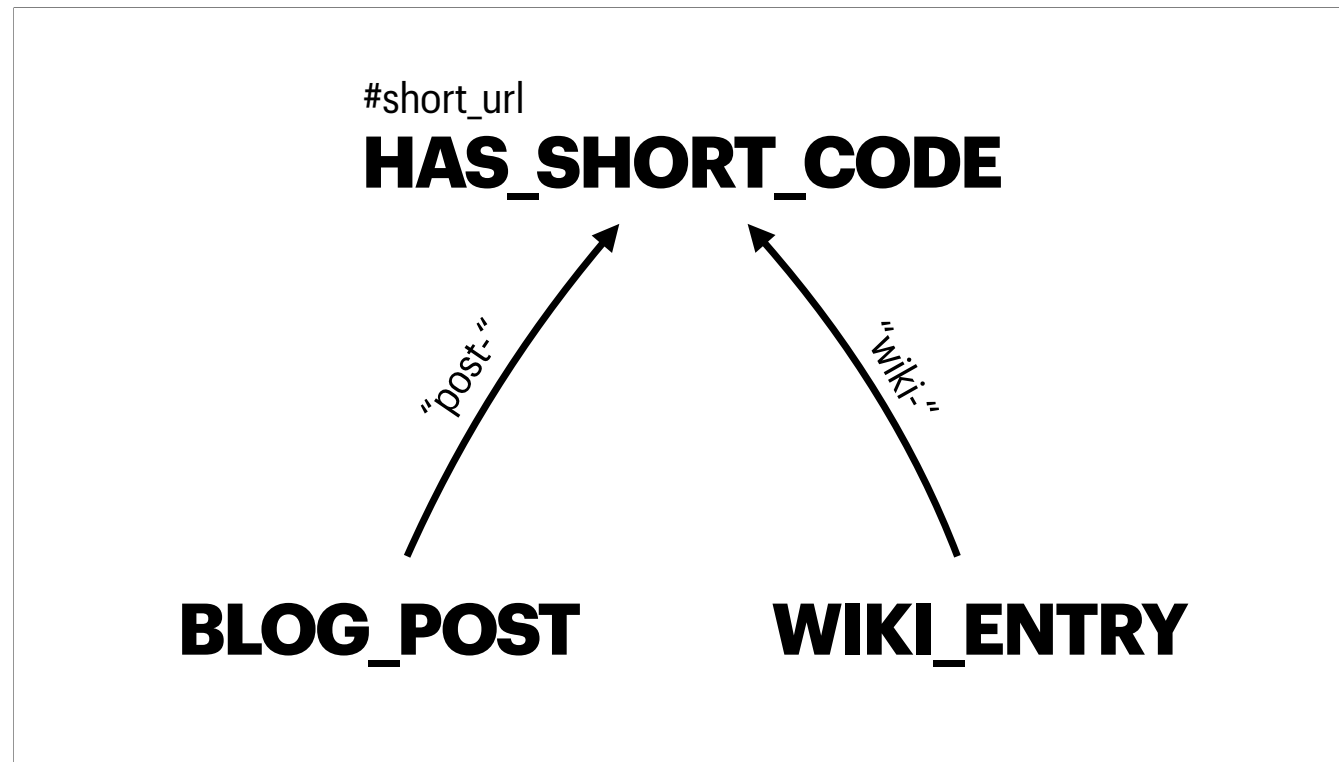
```
module CarWash = struct
  let price_per_wash = 42.

  let wash car =
    print_endline "washing!";
    car
;;
end
```

Then there are modules. Modules are essentially groupings of functions and values to keep similar things in the same place.

As a quick example, here's a small module for describing a Car Wash. The module is named Bubbles, and it contains values like `price_per_wash` for holding the price, and functions like `wash` which takes in a car, outputs a message, and returns the same car.

As a side note, but a very important one - note that these are NOT classes. There are no instances of car wash, the car wash itself has no state, the values on car wash can't be changed, it just is a kinda bucket to hold a collection of loosely related things.



Modules can be included into other modules, and this (just like in Ruby), is one of the primary ways that we avoid duplicating a ton of code for similar features. I might have a module for describing Blog Posts and a module for describing Wiki Pages and want them both to be able to have short URLs. I can introduce a third module that contains those methods and share relevant parts of the implementation.

Many times, when I use this pattern, there are details about the behavior of the module that need to be passed from the class into the module. We might get back a method like `#short_url` from `has_short_code` but we have to pass in class-specific prefixes like `"post-"` and `"wiki-"` to get it to work properly.

Being as `include` doesn't take any arguments, how do we get `HasShortCode` the information it needs?

There are actually two ways that we typically go about solving this shared logic problem in Ruby.


```
module HasShortCode
  def short_url
    "https://example.com/#{self.class.short_code_prefix}#{id}"
  end
end

class BlogPost < ActiveRecord::Base
  include HasShortCode

  def self.short_code_prefix
    "post-"
  end
end
```

The first would be to define methods with known-names on the including class that the module can then assume will be there and call.

On the top here we have the implementation of HasShortCode and on the bottom you can see how it's used. We include has_short_code and then we're expected to define a method called short_code_prefix which returns the prefix to use.

```
class BlogPost < ActiveRecord::Base
  include HasShortCode

  has_short_code prefix: "post-"
end
```

The second, and one you'll see more in Rails but maybe haven't noticed or thought much about how it's working, is to define a method that does this same thing for you, giving a nicer interface to the same behavior that requires defining less methods.

```
ActiveSupport.on_load(:active_record) do
  send :include, HasShortCode
end
```

To make these easier to work with, and avoid the extra include line, many library authors will go and include their module into the AR::Base on load.

This is how you get the typical behavior you're probably accustomed to, where you can install a gem and then start using these one-liners to add behaviors to your classes.

```
module HasShortCode
  extend ActiveSupport::Concern

  module ClassMethods
    attr_reader :short_code_prefix

    def has_short_code(prefix:)
      @short_code_prefix = prefix
    end
  end

  def short_url
    "https://example.com/#{self.class.short_code_prefix}#{id}"
  end
end
```

The implementation of these methods normally will look something like this:
Manipulating the outer class to store the attributes we'll need

So here we're defining a module HasShortCode and when that module gets included, you get two class methods, one to retrieve the short code called short_code_prefix and one to set it called has_short_code. The latter one is the one you're actually calling from inside of your model. Then the implementation of short_url is just the same as in the previous example.

```
p BlogPost.instance_variables  
# [:@short_code_prefix]
```

When us library authors do this, we litter a bit of state around: the actual variable holding BlogPost's short_code_prefix lives directly on the class

```
class Unrelated < ActiveRecord::Base
end
```

```
Unrelated.has_short_code
```

```
Unrelated.short_code_prefix
```

And all of our methods exist on every model, even the ones not using short codes. Some of these can be avoided leaking everywhere, but many libraries don't go through those hoops. A quick look at what's on AR::Base might surprise you!

Of course none of this is the end of the world, but functors provide a potential third approach.

```
module IncX (M : X) = struct
  let x = M.x + 1
end
```

← **FUNCTOR**

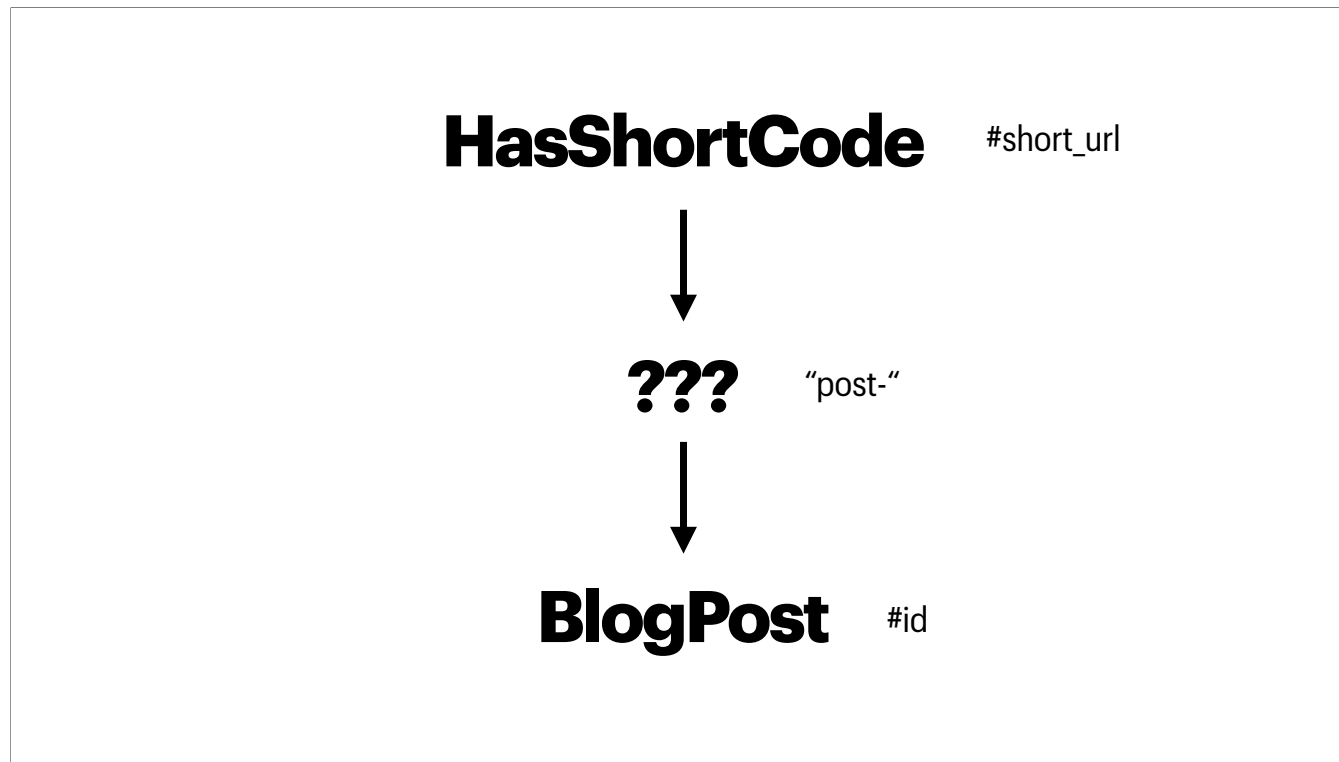
```
module A = struct let x = 0 end
```

```
module B = IncX (A)
```

<https://cs3110.github.io/textbook/chapters/modules/functors.html>

With functors in OCaml, which you can roughly think of as functions from one module to another, we can pass details that modify the behavior of a module at the point we're creating it. Here's an example of a functor that takes as input an existing module with a value `x`, and returns a new module where that value is one higher than before

If we could do the same in Ruby, maybe we could use that behavior to dynamically create a module that was modified in the right way so that when it was included it already knew what the prefix was. Can we do that? Sure we can!



Well, in Ruby modules are just objects, we can create them - so maybe we can go ahead and create another module that sits between the class and HasShortCode at the point of inclusion and put our extra methods and properties there?


```
class BlogPost < ActiveRecord::Base
  include HasShortCode.make(prefix: "post-")
end
```

To give us a solid hook into the point of inclusion, we can define a module-method on HasShortCode. We'll call it `make`. Make will have access to the prefix and due to the way it's called, can also return a new module which will be included into BlogPost.

Let's implement this!

```
module HasShortCode
  def self.make(prefix:)
    Module.new do
      include HasShortCode
    end
  end
end
```

Creating a new module is trivial with `Module.new`, and inside of that module we can actually just include the original module:

So we've not added any behavior here, but we've set up for the thing we want to add because as you can see `make` now has scope on the `prefix` variable. We're close!

```
module HasShortCode
  def self.make(prefix:)
    Module.new do
      define_method :prefix do
        prefix
      end

      include HasShortCode
    end
  end
end
```

Now we just have to define methods on the proxy module for each of the passed in args. There are a few ways to do this but for simplicity's sake, we can use `define_method`.

```
class BlogPost < ActiveRecord::Base
  include HasShortCode.make(prefix: "post-")
end
```

```
p BlogPost.instance_variables
# []
```

```
Unrelated.has_short_code
# NoMethodError
```

```
Unrelated.short_code_prefix
# NoMethodError
```

And boom, we can write a library using this pattern and include it with a single line. We can also demonstrate that the littering of methods doesn't happen here because of the module sitting in between.

No instance variable on BlogPost, and no methods defined on AR::Base and thus not available on other subclasses not using short codes. No more need for an initializer hook even!

But man - It's a shame this code is so specific to HasShortCode, maybe it doesn't have to be? We can extract the concept of the Functor into a separate module easy enough:

```
module Functor
  def make(**kwargs)
    base = self
    Module.new do
      kwargs.each do |name, value|
        define_method name do
          value
        end
      end

      include base
    end
  end
end
```

So we've got a functor module, it's got a method called make that creates a new module, defines methods for each of the kwargs, and then includes the underlying module before returning.

```
module HasShortCode
  extend Functor

  def short_url
    "https://example.com/#{prefix}#{id}"
  end
end
```

So when library authors want to make use of this pattern, they can just extend Functor.

```
module HasShortCode
  module InitMethods
    def has_short_code(prefix:)
      send :include, HasShortCode.make(prefix: prefix)
    end
  end
end

class BlogPost < ActiveRecord::Base
  extend HasShortCode::InitMethods

  has_short_code prefix: 'post-'
end
```

An additional neat thing about this implementation is that you can actually use this new version to enable any of the three patterns I've described.

It'll work as a functor of course. It'll work defining your own methods. It's also relatively trivial to define a method to make the inclusion use this pattern but still feel like a natural Rails API to the user, still without leaking extra methods or variables into AR::Base.

(LAST SLIDE OF SECTION)



This is a view of the main stage downstairs

In OCaml, there's a language feature called variants. You can define a type where the value can be one of several states.


```
type color =  
  | Rgba of float * float * float * float  
  | Rgb  of float * float * float  
  | Hex  of string  
  | Name of string
```

So for example, this is the definition of a Color type. Color can typically be expressed in a variety of ways. So a variant lets us create a single type that can represent all the ways that someone might specify color.

Here the value can be an Rgba value which needs carries 4 floats, an Rgb value which carries 3 floats, a Name which must have a named color value, or Hex which must contain a string.

How would we do this in Ruby??

```
class Color
  def initialize(r, g, b, a)
    # ...
  end

  def self.of_rgb(r, g, b)
    new(r, g, b, 1.0)
  end
end
```

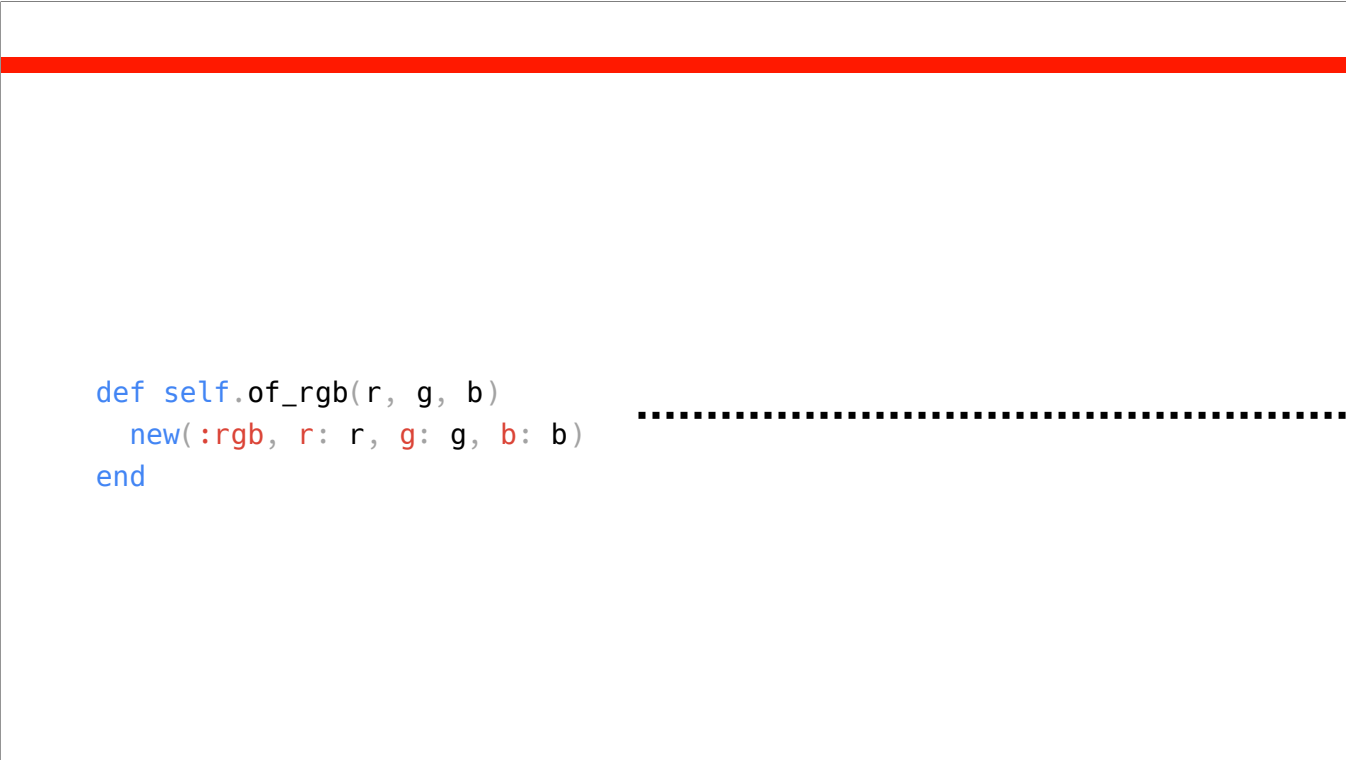
Well, if it was me I'd create a Color class and give it factory methods for each of the individual modes. So here we have a class that can only handle Rgba colors and we use factory methods like `of_rgb` to express the other types in terms of rgba. Some of these are going to be a little complicated, but whatever

In a lot of cases this is the right answer. But it's not quite the same as the OCaml implementation because we've essentially thrown away the original input the user gave before we did the normalization. What if we needed to be able to maintain a user interface that let us show the Color as it was input by the user?

In order to do this we need to store the information as the user entered it so it can be retrieved. So the new problem is how can we represent and store a color that can be in any of these formats



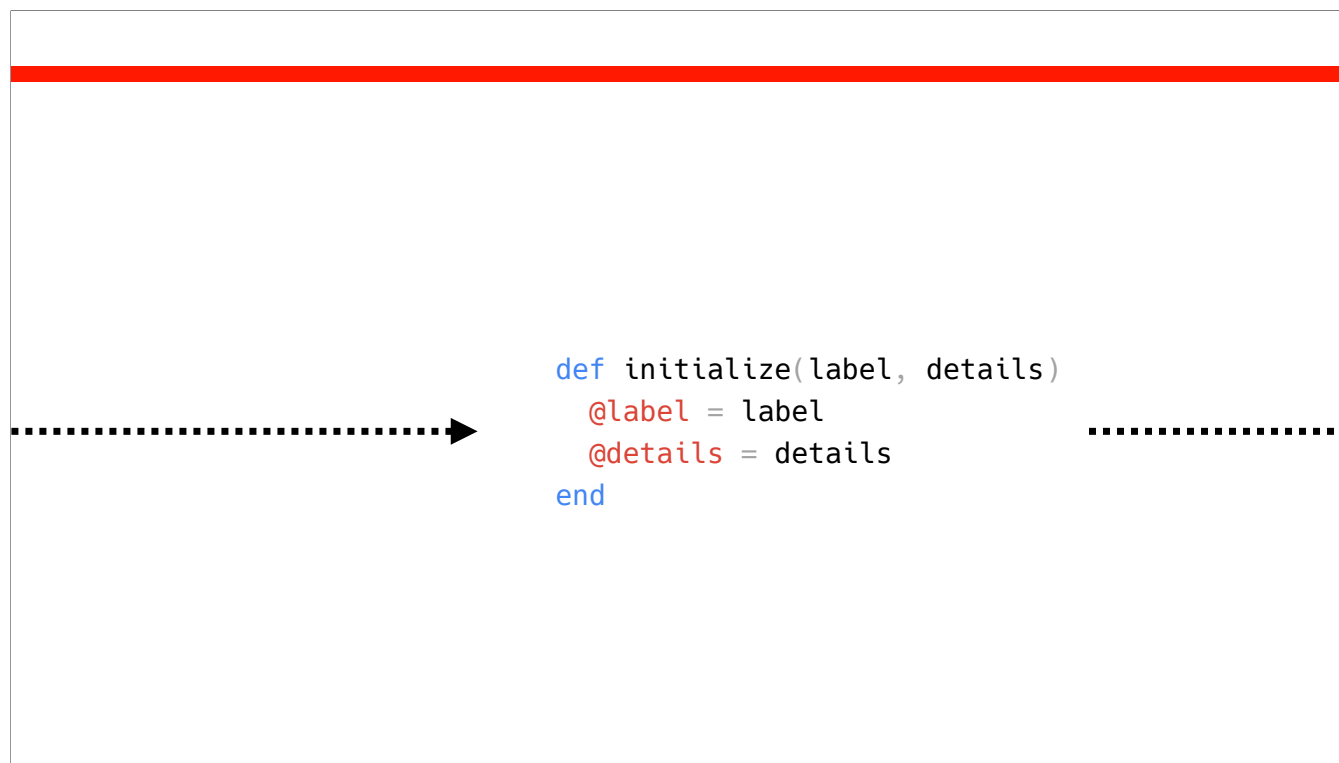
I hope we can all agree that single-table inheritance is a bit overkill for this problem. Instead we'll want to be able to put the color information in a single database column. How can we do that without it getting really complicated dealing with all the edge cases?



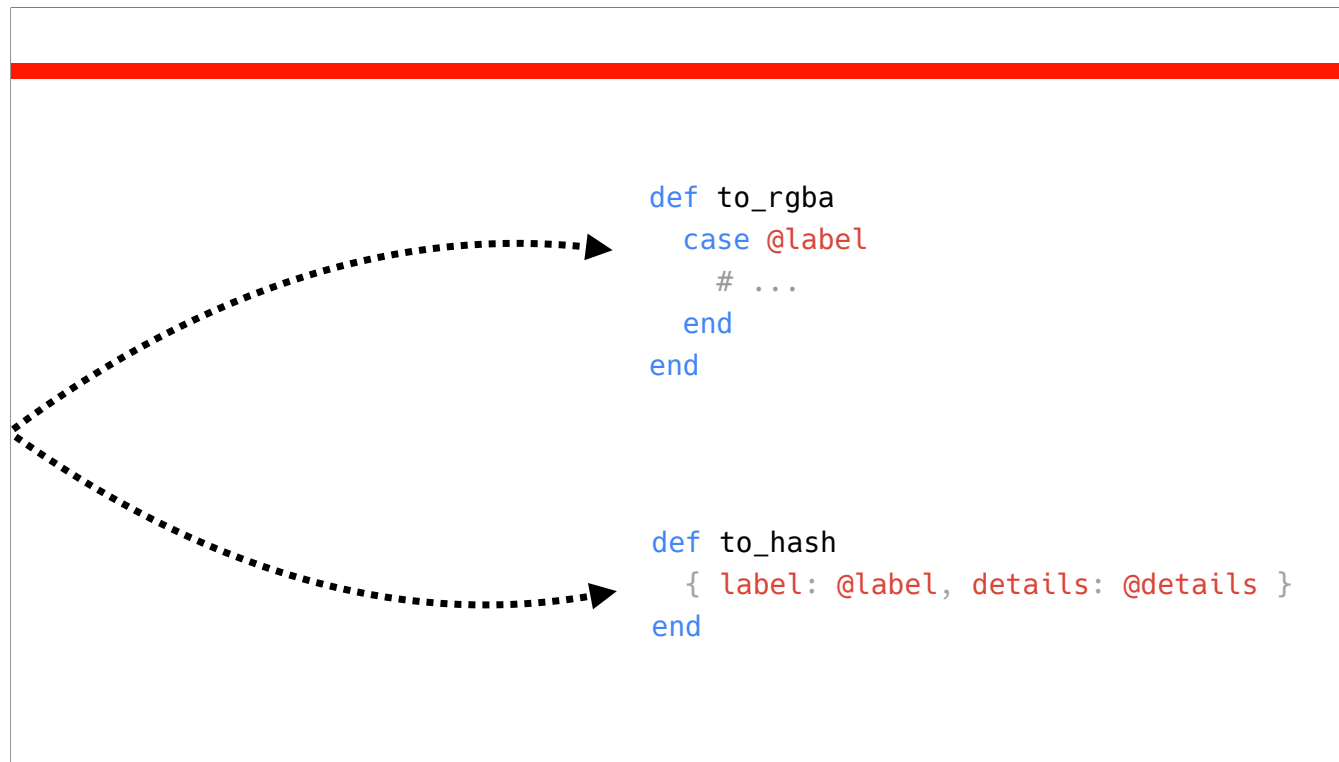
```
def self.of_rgb(r, g, b)
  new(:rgb, r: r, g: g, b: b)
end
```

The diagram consists of a light gray rectangular box. At the top of the box is a solid red horizontal bar. Below this bar, the Ruby code snippet is displayed. A horizontal dotted line extends from the right side of the code block across the width of the box.

Instead, we can modify our Color class to hold the details of the original input and determine the normalized color only when we're actually going to use it to color something. So this would look something like this. Things start with the factory methods, and rather than just pass r, g, b we also pass a label identifying the type of color



Then in initialize, we take in both the type and details



then we implement methods like `to_rgba` and `to_hash`. `to_rgba` does the actual normalization to color things on the user end, and `to_hash` provides us a version we can push into a serialized database field.

```
def initialize(label, details)
  @label = label
  @details = details
end
```

And with this we've actually mostly implemented variants. A value of the Color variant is nothing more than the format (in this case the symbols :rgba or :rgb), and the associated data.

... one annoying thing is that it's shockingly easy to make an invalid color by calling `new` with some random type

But we can fix that in a crude way by just making `new` private

```
# Create a variant
Color = Variant.create(:rgba, :rgb, :hex, :name)

# Create an instance of a variant
Color.new(:hex, r: 255, g: 255, b: 255, a: 1.0)
```

To formalize this pattern a bit more, we can wrap the whole concept up in a tiny library that makes it easy to create and use variant types, and ditches the need to create new factory methods for each type. The way I think we'd want something like that work is like this: To create a new variant, we use `Variant.new` which would return a new class. Then to create an instance we can just call `new` on that new class

```
module Variant
  def self.create(*labels)
    klass = Class.new(P)
    klass.valid_labels = labels.to_set
  end
end
```

And the implementation is pretty straightforward. In the create method we can dynamically generate a new Class and set the labels onto it. When we dynamically create a new class in Ruby (click), we're actually able to specify its parent class, and that's really convenient because that gives us a perfect opportunity to have each of the created Variant classes share common behavior.

```
class Color < P  
end
```

So when you create a new Variant, the class you receive back will new a new class, which descends from P

```
InvalidLabel = Class.new(StandardError)
```

```
class P
  class << self
    attr_accessor :valid_labels
  end
end
```

```
def initialize(label, *args, **kwargs)
  raise InvalidLabel unless self.class.valid_labels.include?(label)
  @label = label
  @args = args
  @kwargs = kwargs
end
end
```

In this case the behavior we share is the initializer of P which let's us hook into the creation of a variant instance and raise on any creations that aren't valid for the given Variant type. That code is rightttttt.. here!

Pretty neat!

```
match c with
| Rgba (r, g, b, a) -> sprintf "rgba(%f, %f, %f, %f)" r g b a
| Rgb (r, g, b) -> sprintf "rgb(%f, %f, %f)" r g b
| Hex v -> v
| Name v -> v
```

Because OCaml is typed they're actually able to go a bit further with variants. The language supports a feature called pattern matching, so you write this which essentially is like a case statement for all the different ways the variable could hold data.

And, when you use this pattern matching on variants, the compiler will actually check to make sure you've handled every possible state the variant can be in.

```
let normalized_color =  
  match c with  
    (* Rgba is missing *)  
  | Rgb (r, g, b) -> sprintf "rgb(%f, %f, %f)" r g b  
  | Hex v -> v  
  | Name v -> v
```

```
File "sum.ml", lines 12-16, characters 4-17:  
Error (warning 8 [partial-match]): this pattern-matching is not exhaustive.  
Here is an example of a case that is not matched:  
Rgba (_, _, _, _)
```

If you don't account for every type of color, the code won't compile. The compiler is stopping you from making an assumption, in this case that not handling a certain case means it has no defined behavior.

The OCaml compiler HAS to have this feature actually. Remember when I said that each variable has to have a known type at compile time and that type can't change?

This means that each branch of a match statement by necessity has to return the same “type” of object. If each of these branches returns a string and then we have an unhandled case, what would the value of `normalized_color` be? In Ruby we might say `nil`, but `nil` has a different type than String. That breaks our invariant!

```
match c with
| Rgb (r, g, b) -> sprintf "rgb(%f, %f, %f)" r g b
| Hex v -> v
| Name v -> v
| _ -> "missing"
```

If you want to leave out cases in an OCaml match statement, you must do that explicitly using a wildcard match and that wildcard match branch has to return the same type as all the other branches.

(this slide intentionally left blank)

Now in Ruby we obviously can't make the code not compile if a case is missing, but can we get something like exhaustiveness checks to help us avoid adding new states for a given type but not accounting for it around our application?

Well this would be a pretty boring talk if the answer was no, so yeah I think we can

```
result = variant.match(  
  one: -> { 1 },  
  two: -> { 2 }  
)
```

The interface we're going to shoot for looks something like this. We'll add a method on all variant types, and then pass it a hash where the key is the type label, and the value is a proc to execute in the case the value holds that type


```
def match(handlers)
  missing_cases = self.class.valid_labels - handlers.keys
  handles_all_cases = missing_cases.none?

  unless handles_all_cases
    raise UnhandledMatchCase, missing_cases.join(', ')
  end

  handlers.fetch(@label).call(*@args, **@kwargs)
end
```

And the implementation is actually really easy. When match is called, we look at the cases that are defined and subtract them from all the valid cases for the variant type.

```
def match(handlers)
  missing_cases = self.class.valid_labels - handlers.keys
  handles_all_cases = missing_cases.none?

  unless handles_all_cases
    raise UnhandledMatchCase, missing_cases.join(', ')
  end

  handlers.fetch(@label).call(*@args, **@kwargs)
end
```

If there are any cases left, that means we're missing a case and thus we raise an error

```
def match(handlers)
  missing_cases = self.class.valid_labels - handlers.keys
  handles_all_cases = missing_cases.none?

  if !handles_all_cases && !handlers.key?(DEFAULT_KEY)
    raise UnhandledMatchCase, missing_cases.join(', ')
  end

  handlers.fetch(@label, handlers[DEFAULT_KEY]).call(*@args, **@kwargs)
end
```

Now we also need to be able to handle that default case, to do that we can just avoid raising in the case that a default key is defined, and fall back to the default key if a given handler isn't found

```
def match(handlers)
  missing_cases = self.class.valid_labels - handlers.keys
  handles_all_cases = missing_cases.none?
  extra_cases = handlers.keys.to_set - self.class.valid_labels - [DEFAULT_KEY]

  if !handles_all_cases && !handlers.key?(DEFAULT_KEY)
    raise UnhandledMatchCase, missing_cases.join(', ')
  end

  raise ExtraMatchCase, extra_cases.join(', ') if extra_cases.any?
  raise UnnecessaryDefaultCase if handles_all_cases && handlers.key?(DEFAULT_KEY)

  handlers.fetch(@label, handlers[DEFAULT_KEY]).call(*@args, **@kwargs)
end
```

Lastly, because why not, we're here and OCaml does it - let's implement two more errors. One for the case that the match statement handles cases that aren't relevant, and one where a default is provided but will never be used because the other options are exhaustive already.

VARIANT CASES

- **Cases exist but are not handled**
- **Cases are handled but don't exist**
- **Instances are created of invalid cases**

So with this interface, you'll see errors at locations where

cases exist but are not handled, common when you're adding a new case to an existing variant

cases are handled but don't exist: common when either removing an existing case, or a developer has made a typo on a case

instances are created of invalid cases: common when mixing up types, typoing names, or removing cases without removing the matching logic from around your app

```
1) Variant raises if trying to match without all cases
Failure/Error: raise UnhandledMatchCase

Variant::UnhandledMatchCase:
  one, two
# ./var2.rb:33:in `match'
# ./var2.rb:89:in `block (2 levels) in <top (required)>'
```

If you use this pattern or something like it in parts of your Rails application, and have existent but maybe spotty test coverage, you'll be able to change the underlying variant cases confidently knowing that your tests will fail if cases are not handled appropriately.


(LAST SLIDE OF SECTION)



Now we're going to discuss the last feature for today: Monads

Fast forwarding through any inappropriate jokes, I've gotta come clean on a little fib I kinda told earlier.

```
let normalized_color =  
  match c with  
  | Hex v -> v    (* String *)  
  | Name v -> v   (* String *)  
  | _ -> null     (* Null? *)
```



mismatch!

I had said that the wildcard case of a pattern match couldn't return `null`, because null is a different type than String.

In a lot of other statically typed languages, like Java, that's not really true!


```
String str = null;
```

In Java, null isn't a type, but instead a value that can be any type to indicate absence. This statement makes sense in Java:

```
String str = null;
```

The value is null, but the type is definitively String.

This will feel familiar to Ruby developers, albeit that the actual mechanism is slightly different. In Ruby: String and nil are different types, in Java, null is just “the null value” of all types, a special value that exists to indicate absence.



~~null~~

The real reason that the wildcard case in OCaml can't return null, is that null doesn't exist as a concept in OCaml. It's not named a different thing, it just doesn't exist.

BILLION \$ MISTAKE ?

- **No compile-time guarantees**
- **Meaning has to be known on both sides**

In some circles, Null in general is fairly contentious, it's worth quickly running down why some people aren't super into it:

No compile-time guarantees around handling
Meaning has to be known on both sides

For these reasons, OCaml has skipped over the idea of null entirely, but often we still need something like it. That is to say: we need the idea of a variable that can hold a value like an Integer, but can also hold no value. This is a case where monads can come to the rescue, in particular a monad referred to as the 'Option' or 'Maybe' monad..

```
type option =  
  | None  
  | Some of 'a
```

Monads in OCaml are built using variants. And the Option monad type has this shape: (None | Some 'a). So this is a type that can either have the label 'None' in the case it's representing an empty value, or 'Some' in the case it's representing a value.

```
let x : int = 41  
let x_op : int option = Some 41  
let n_op : int option = None
```

And just like we saw before with Color, these Option monads have their own type. So while 42 is an 'Int', (Some 42) is an 'int option'.

The upshot of this is that when a function takes in an int that is optional, it actually takes in a parameter of the type 'int option'. There are a variety of ways to turn this 'int option' back into an 'int'.....

```
let _ = match n with
| Some value -> sprintf "n: %d" value
| None -> "no value" in
```

but the most common one is a match statement. Here we have a match statement for handling an option, and you can see that we have to explicitly handle the Some and None cases in each match to satisfy the exhaustiveness checker.

So these are similar to Null values, with a little bit more work in the average case, but safe from mistakenly introducing null pointer exceptions.

Because we have to handle everything that is known to potentially be null, you'll never have code that thinks something shouldn't be null when it is.

```
module Option
  Value = Variant.create(:none, :some)
end
```

Building something like this within the framework of variants I described previously is really straightforward. We'll create an Option module to wrap everything, and inside a variant called Value which can either be none or some.

But this is just a variant, it's not a monad yet. Monads also have this idea of composability. We typically talk about monads as having an interface with two methods return & bind. Let's find out what they do and implement them in Ruby

```
def self.return(value)
  if value.nil?
    Value.new(:none)
  else
    Value.new(:some, value)
  end
end
```

Return is easy it's a method that takes in a value and returns a monad. In the case of the option monad, that means just returning the value wrapped in the variant. Because this Ruby and we kinda don't hate null as much as some other people, we'll consider a null value to be :none and other values to be :some


```
val bind : 'a t -> ('a -> 'b t) -> 'b t
           'a option      'a → 'b option      'b option
```

The other method, bind is a little bit more complicated but not by much. Bind for a monad should have this signature. This roughly reads as: take in a monad, and a function that takes in non-monad value and returns a new monad. You can think of bind as unwrapping a monad into some context. Unwrapping can mean different things for different monads, but for the option monad unwrapping means 'do nothing if this value is none, otherwise, take out the value and pass it in'

```
def self.bind(op)
  v = op.match(
    some: ->(v) { yield v },
    none: -> { }
  )

  self.return v
end
```

So to write that implementation in Ruby, we can build again on match statements and write it like this:

```
n_op = Option.return 42

Option.bind n_op do |n|
  p n # 42
end
```

And this works! We can now create options and unwrap them using bind.

```
Option.bind x_op do |x|
  Option.bind y_op do |y|
    p x + y
  end
end
```

```
x_plus_one_op = Option.bind x_op do |x|
  x + 1
end
```

We can compose multiple monadic operations together easily, and execution will stop after the first one that results in a None value.

We can also convert one option value into another without taking it out of its box

```
if x && y
  p x + y
end
```

```
x&.abs
```

Those previous two examples have less safe analogs you definitely already know. Namely IF statements and the safe navigation operator.

The key difference here is that the safe-navigation operator is possible for users of your APIs to ignore. It's easy to get code that runs in the happy paths and doesn't error without accounting for the fact that something can be null.

This option monad pattern on the other hand, the developers using APIs designed this way HAVE to contend with the option by explicitly unwrapping it. You shouldn't use it everywhere, but in cases where not considering null could have disastrous consequences, it's something you might consider.

RESULT

SEARCH

STATE

IO

IDENTITY

LIST

It's worth checking out other common monads as well, each one is part of a design pattern that communicates more information to users of a given API what to expect when dealing with a response.

(LAST SLIDE OF SECTION)



I feel like I have this point in my talk every time, so let's just BREATHE

This was a lot! We went from maybe not knowing that ocaml existed, to learning about how it differs from Ruby, to tearing some functionality back over the line. I hope I've convinced you if nothing else that there's just an INFINITE amount of amazing stuff out there to learn.

A lot of things we see as language features are actually just design patterns, and in a language as syntactically flexible as Ruby, we can make them feel really natural.



Go and learn a functional language, learn a logic language, declarative language, whatever language - not because you might switch or because you're moving on, but because it might make you a better developer. Find ways to bring those ideas back to our community and ensure that this language and framework can live for another 15 years.

Thank you and I hope you enjoy the closing keynote.

SEEJOHNRUN

