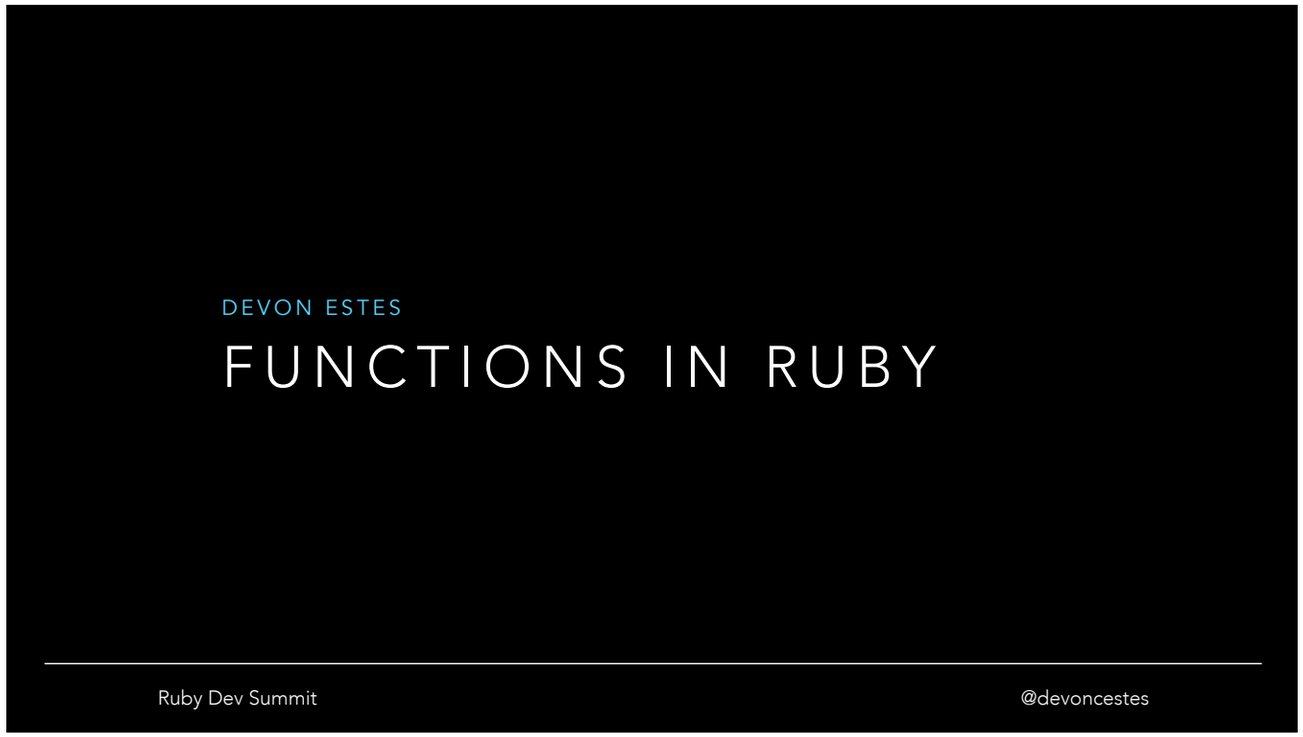


I bet if you ask 100 Ruby developers what their biggest source of frustration is on a day to day basis, one thing would be brought up again and again.

Nobody likes legacy code. It’s often obtuse, complex, and difficult to work with. And the worst part of all is it doesn’t make you happy. Working with Ruby is supposed to be about developer happiness, but there’s no joy in working with legacy code.

```
class Calculator
  def add(first, second)
    first + second
  end
 end
```


The tough thing about legacy code is it sneaks up on you. You start off with nice tiny methods on nice tiny classes, and the next thing you know, your methods looks like this.

```
class Calculator
  def add(first, second, print_results, write_to_file, print_to_stderr)
    result = first + second
    if print_results
      puts result
    elsif write_to_file
      File.write("math.txt", result)
    elsif print_to_stderr
      $stderr.puts(result)
    end
    result
  end
 end
```


This is an unfortunately common predicament. And today I'm going to offer one idea that might sound a little weird to some, and less weird to others, but let's just take this ride together. I think once we get to the end you'll be convinced.

I think we need to be using more functions. In Ruby.

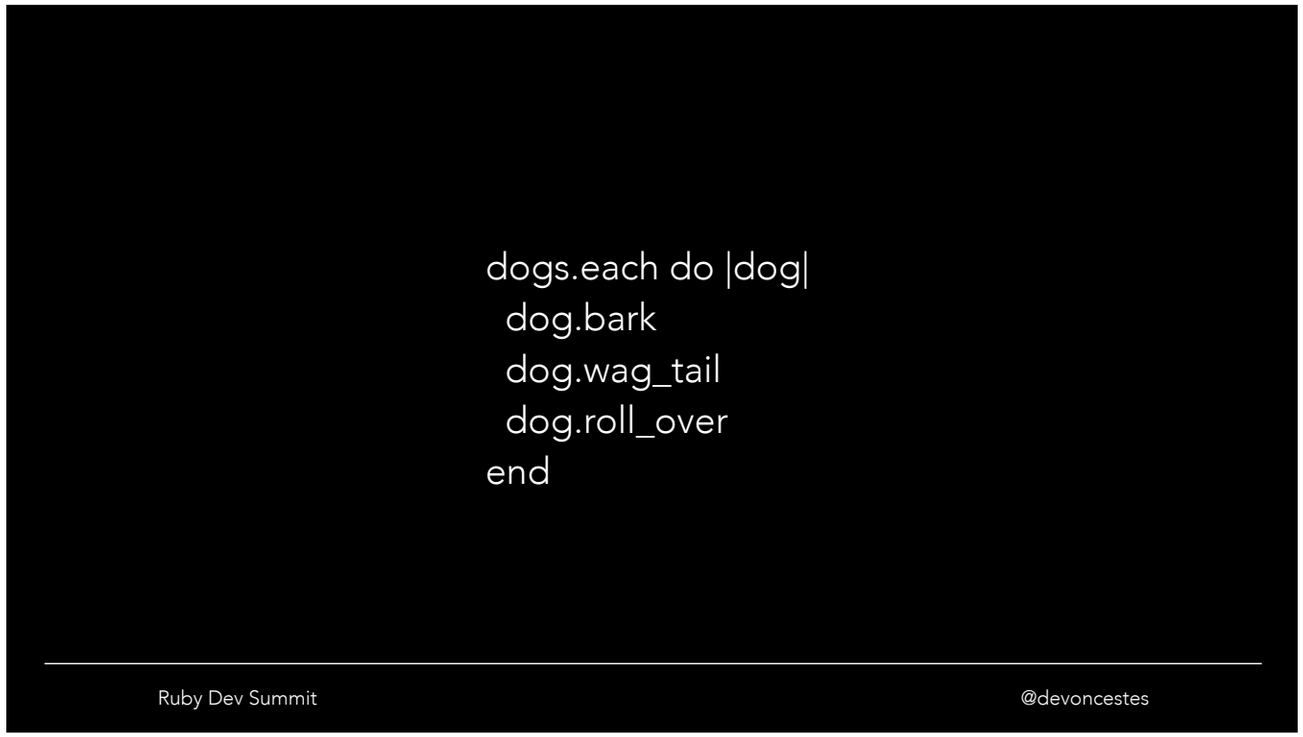
But Devon, you're saying, there are no functions in Ruby! Everything's an object!

EVERYTHING IS *NOT* AN OBJECT

Ruby Dev Summit

@devoncestes

There is one thing in Ruby that isn’t an object.



I'll give you a hint. Somewhere in this code, we have something that is crucial to Ruby, but is in fact not an object.

Have you spotted it yet?

```
dogs.each do |dog|
  dog.bark
  dog.wag_tail
  dog.roll_over
end
```


That’s right. The block - one of Ruby’s most powerful inventions.

BLOCKS ARE FUNCTIONS

Ruby Dev Summit

@devoncestes

Blocks are very much not objects. You can’t instantiate an instance of the Block class. It has no state, and only behavior. In fact, when you look at the C code in the MRI implementation that most of you are probably using on a day to day basis, you’ll see that blocks are indeed very special.

CLOSURES

Ruby Dev Summit

@devoncestes

Blocks are closures, which is a very old idea taken from Lisp in the 70’s. But you don’t need to take my word for it!

Matz has himself described how Lisp was an inspiration for blocks and for the type of higher order functions that we've now come to take for granted as one of the most powerful pieces of Ruby.

```
dogs.each do |dog|
  dog.bark
  dog.wag_tail
  dog.roll_over
end
```


But blocks aren’t the only functions in Ruby! There are several other function-like objects that we have at our disposal. Let’s look at a couple different ways we can write this snippet of code here.

```
dog_fun = ->(dog) do
  dog.bark
  dog.wag_tail
  dog.roll_over
end

dogs.each do |dog|
  dog_fun.call(dog)
end
```


We can use a lambda. A lambda, as its name implies, is very much a function.

```
dog_fun = ->(dog) do
  dog.bark
  dog.wag_tail
  dog.roll_over
end

dogs.each do |dog|
  dog_fun.call(dog)
end
```


Sure, in this case `dog_fun` points to an instance of the Proc class (which is another function. For today we're just going to gloss over the differences there because they're not important to this discussion), but what we really have is a function.

```
module Dog
  def self.call(dog)
    dog.bark
    dog.wag_tail
    dog.roll_over
  end
end

dogs.each do |dog|
  Dog.call(dog)
end
```

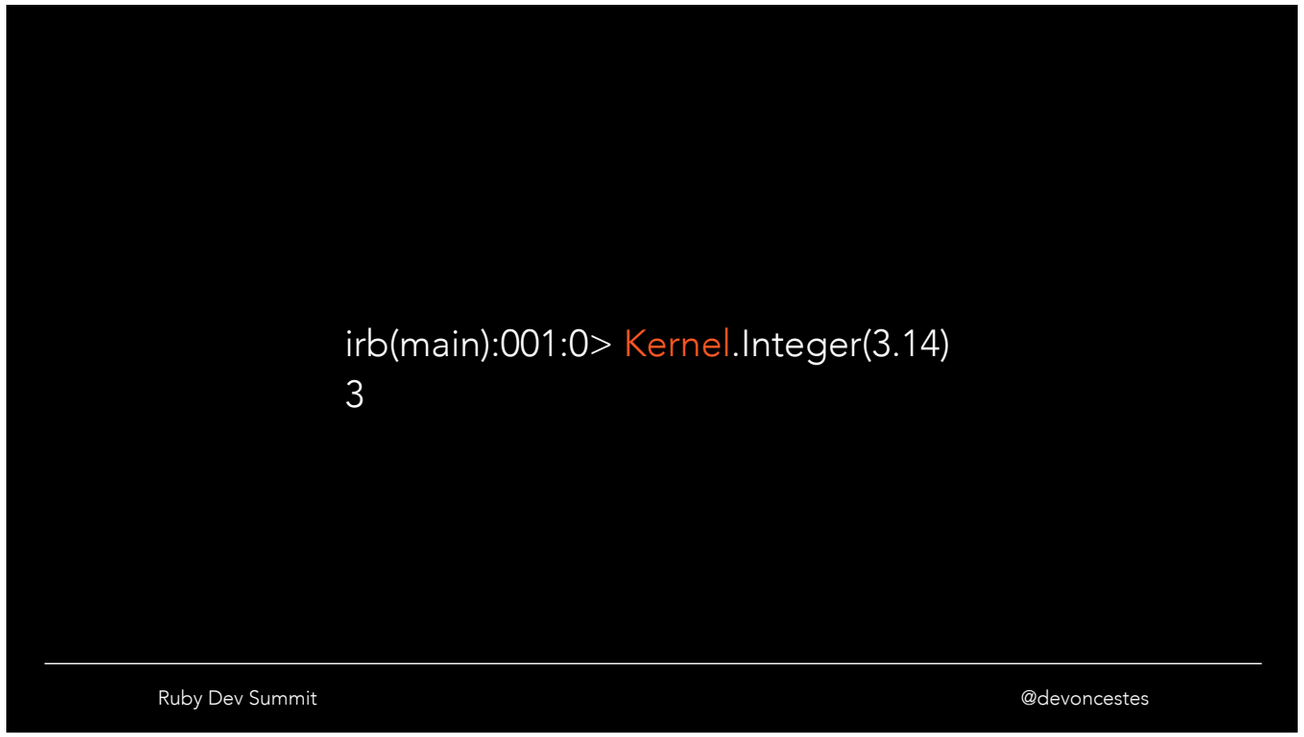

We can also define a module function. So, `Dog` here is an instance of the the Module class, which makes it an object. But this object responds to `call` in the same way a lambda or proc does.

```
module Dog
  def self.call(dog)
    dog.bark
    dog.wag_tail
    dog.roll_over
  end
end

dogs.each do |dog|
  Dog.call(dog)
end
```


I know you’re thinking to yourself that this is crazy. Who would do this? Well, instances of module functions can be found all throughout the Ruby standard library.

Have you ever seen this? It's weird, I know. A capital I there, but it has parentheses. It's not a class that we're instantiating, it's not a method. What it is, is a function! It's actually a special type of function called a conversion function. The way I've written it here doesn't tell the whole story, though.



You can also write it like this. So, Integer is a method defined in the Kernel module. Because Kernel is included in Object, all these functions are available pretty much everywhere, but using the fully qualified name here is also totally valid.

WHAT MAKES A FUNCTION?

- Accepts 0 or more inputs
- Returns 0 or more outputs
- Does not have (or use) persistent state
- Can access state in any outer scope (closures!)
- Can have side effects

So, let’s go over a bit about what we’ve learned about functions in Ruby based on looking at some of those examples in the language itself.

WHAT MAKES A METHOD?

- Uses or mutates the internal state of the receiver
- That's it. Really.

So the difference? It's actually quite small.

So, as we've seen, there are some methods that can behave like functions. Those methods might be defined on an instance of a class, but because they don't use or mutate any of the internal state of the message receiver, they actually behave like functions.

```
class Calculator
  def initialize(name)
    @name = name
  end

  def name
    @name
  end

  def add(first, second)
    first + second
  end
end
```


Let’s show a good example of the difference here.

```
class Calculator
  def initialize(name)
    @name = name
  end

  def name
    @name
  end

  def add(first, second)
    first + second
  end
end
```

Method!

Internal State!

```
class Calculator
  def initialize(name)
    @name = name
  end

  def name
    @name
  end

  def add(first, second)
    first + second
  end
end
```

Function!

So if you ever decide to spend some time in the functional programming world, eventually you'll hear someone talk about pure functions. They're kind of important, so we'll cover that briefly here.

WHAT MAKES A FUNCTION?

- Accepts 0 or more inputs
- Returns 0 or more outputs
- Does not have (or use) persistent state
- Can access state in any outer scope (closures!)
- Can have side effects

If we go back to our checklist for what makes a function, pure functions are really quite similar. They can do everything, except have side effects.

WHAT MAKES A FUNCTION?

- Accepts 0 or more inputs
- Returns 0 or more outputs
- Does not have (or use) persistent state
- Can access state in any outer scope (closures!)
- ~~Can have side effects~~

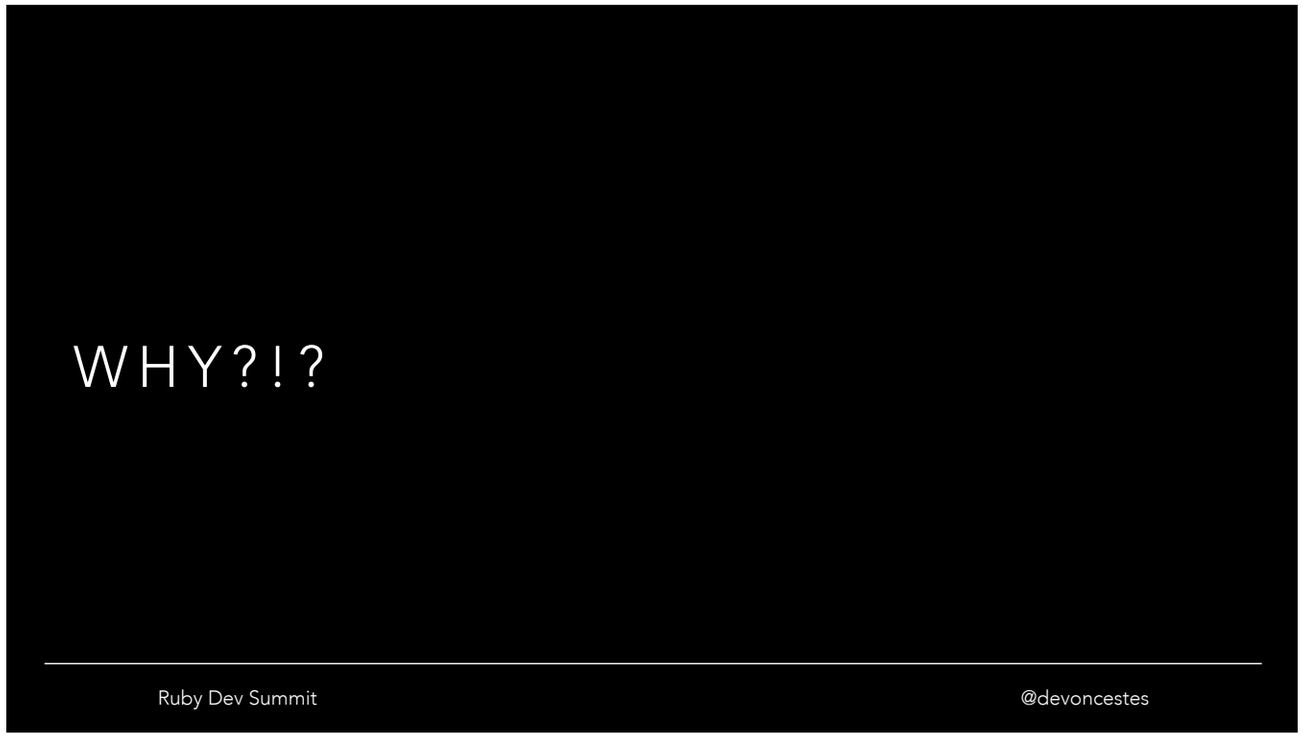
What's a side effect you ask? Well, basically a side effect is something that happens outside of the scope of the function. If you're reading from a file, touching a database, making any sort of network call, or mutating any state outside of the local scope of the function you're in, that's a side effect.

```
class Calculator
 def add(first, second)
   puts “Adding #{first} to #{second}”
   first + second
 end
end
```


So, let’s just show a quick example. Here, we have a side effect because we’re sending output to standard out.

```
class Calculator
 def add(first, second)
  puts "Adding #{first} to #{second}"
  first + second
 end
end
```

Side Effect!



So what’s the point of all this? In short, it’s all about keeping things simple.

The function is pretty much the simplest abstraction we have in all of computer science. It’s a concept even elementary school kids can fairly easily comprehend. Its application to solving problems in code is where things start to get tricky, but a single function on its own is exceedingly simple. And we can use these functions to help keep our applications from getting too complex. That’s where their real power is, and that’s what we’re going to cover now.

```
class Calculator
  def add(first, second, print_results, write_to_file, print_to_stderr)
    result = first + second
    if print_results
      puts result
    elsif write_to_file
      File.write("math.txt", result)
    elsif print_to_stderr
      $stderr.puts(result)
    end
    result
  end
 end
```


So who remembers our legacy code example from way at the beginning of this talk? Pretty gnarly, right? Of course I wrote this to be about as bad as possible, but I promise you that at least a few times in your career you either have seen, or will see, something worse than this. This method now has a ton of complexity, but we can make it much, much simpler thanks to functions!

```
class Calculator
  def add(first, second)
    result = first + second
    yield result if block_given?
    result
  end
 end

Calculator.new.add(1, 2) { |result| puts result }
Calculator.new.add(1, 2) { |result| File.write("math.txt", result) }
Calculator.new.add(1, 2) { |result| $stderr.puts(result) }
```


If we strip out all those conditionals, we can then pass a block to handle any of the side effects that we want for this function. We now have essentially infinite flexibility without having to change the original method. This is basically the textbook definition of code that's open for extension and closed for modification. So, we have arrived at much better object oriented design through the use of functions! Pretty crazy, huh?

So, we now have a function, that accepts another function as an argument, and it's exceedingly well designed in terms of object orientation.

```
RSpec.describe Calculator do
  it “returns the sum of the two numbers” do
    expect(Calculator.new.add(1, 2)).to eq 3
  end

  it “yields the sum to the block if given” do
    num = nil
    Calculator.new.add(1, 2) { |result| num = result }
    expect(num).to eq 3
  end
end
```


One of my favorite things about using functions is how easy they are to test - especially if they’re pure functions. For pure functions, the only thing you need to test is the output. For functions that accept another function as an argument - be that a block, or some other function like argument - you just need to test that the function was called with the appropriate arguments. And for functions with side effects, you do still need to test each of those side effects.

```
class Calculator
  def add(first, second, extra_fun = ->(*_) {})
    result = first + second
    extra_fun.call(result)
    result
  end
 end
```


And just to show another way in which we can write that exceedingly flexible function, here we've replaced the block with a lambda.

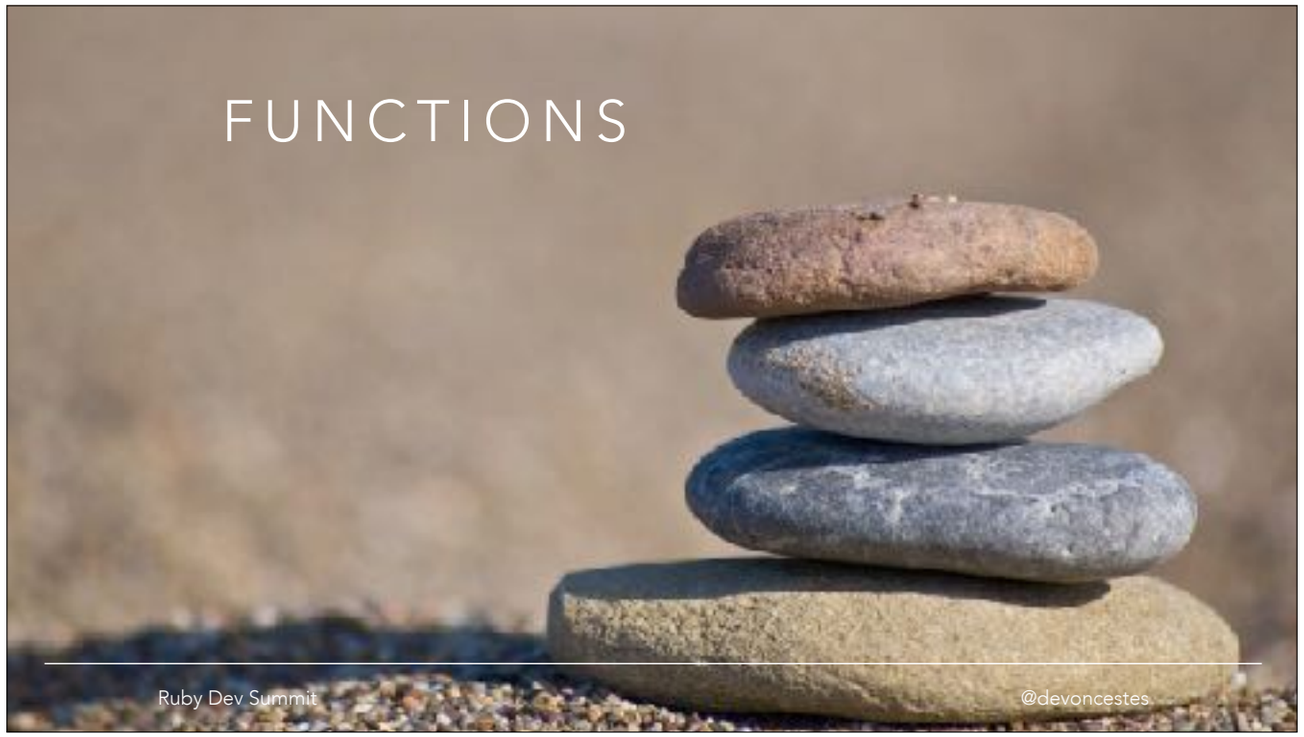
```
class Calculator
 def add(first, second, extra_fun = ->(*_) {})
  result = first + second
  extra_fun.call(result)
  result
 end
end
```


We've also been able to replace that conditional, and we now have a benign value in as our extra function by default. That lambda that we've defined there is a sort of universal NOOP. It accepts any number of arguments, does nothing with them, and returns nil. So, it's totally harmless.

"In object-oriented solutions, small, interchangeable objects collaborate by sending messages. Messages afford seams which allow you to replace existing objects with new ones that play the same role. Message sending makes it easy to change behavior by swapping in new parts."

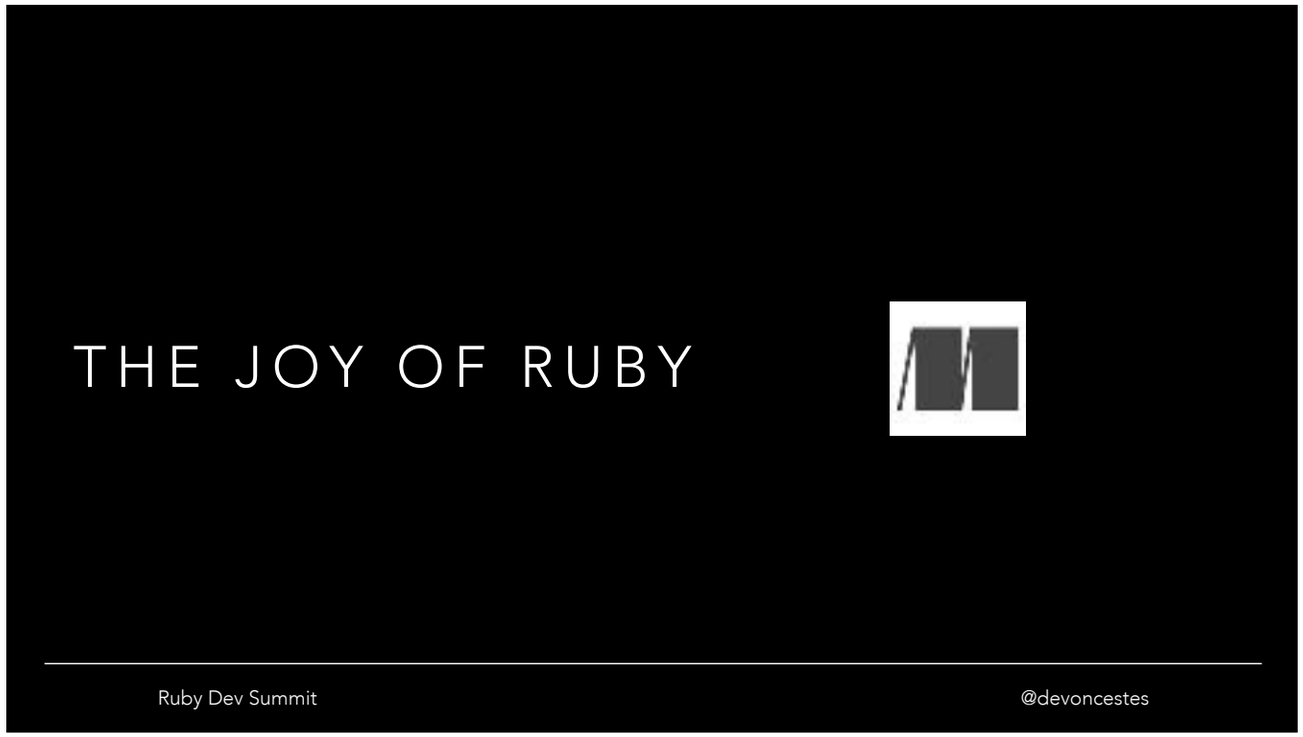
- SANDI METZ

Think about this in regards to the example we just looked at. We have an object, and we send a message to that object. That's all we're doing, really. Sure, it's a very functionally inspired



So, as crazy as it may seem, functions are a really great way to reduce complexity in a Ruby application. They're small and easy to replace in just about any context without complexity.

What we've covered here today is really just the tip of the iceberg. A bird's eye view. If you're still curious, I'm actually writing a book on this topic, which will cover what I spoke about today in much greater detail.



It’s called The Joy of Ruby, and it will be published by Manning sometime in the latter half of 2018. If you’d like to get a reminder about the book when it’s ready - and probably with a discount code and maybe early access to a chapter or two - you can sign up at my website.

DEVONESTES.COM/FIR