# Cool Functional Tricks

# In Ruby

Julian Doherty
@madlep
juliandoherty.com


# We're hiring

careers.envato.com

# Who here has written code like this?

```ruby
def calculate_damage(current_damage, new_damage)
  current_damage + new_damage
end

calculate_damage(5123, 23145)
# 28268
```

# Or this?

```
movies = [
  {movie: "Apollo 13", year: 1994, damage_cost: 2600000000},
  {movie: "Cast Away", year: 2000, damage_cost: 60000000},
  {movie: "Sully", year: 2016, damage_cost: 101000000}
]

total_damage_cost = movies.map{|movie|
  movie[:damage_cost]
}.sum()
# 2761000000
```

# Or this?

```
movies = [
  {movie: "Apollo 13", year: 1994, damage_cost: 2600000000},
  {movie: "Cast Away", year: 2000, damage_cost: 60000000},
  {movie: "Sully", year: 2016, damage_cost: 101000000}
]

total_damage_cost = movies.reduce(0){|total, movie|
  total + movie[:damage_cost]
}
# 2761000000
```

# Or even this?

```
movies = [
  {movie: "Apollo 13", year: 1994, damage_cost: 2600000000},
  {movie: "Cast Away", year: 2000, damage_cost: 60000000},
  {movie: "Sully", year: 2016, damage_cost: 101000000}
]

total_damage_cost = movies.sum{|movie| movie[:damage_cost]}
# 2761000000
```

# What do we call this kind of code?


We call it...
Tom Hanks
Trashing a whole bunch of stuff


The Little Reader's Collection of Children's Stories
REAL REAL BIG TRUCKS!


US AIRWAYS

# TDD

...

# TDD

## Tom Driven Destruction

# Let's figure out how much.

# But let's not make the problem worse

# How do we minimise the damage?

# Three Things

→ Statelessness

→ Immutability

→ Functional Purity

# Statelessness

Code that is in it's own little world.
All it has, is what is given to it.
**There is nothing outside. There is no state**

# Don't do this 😱

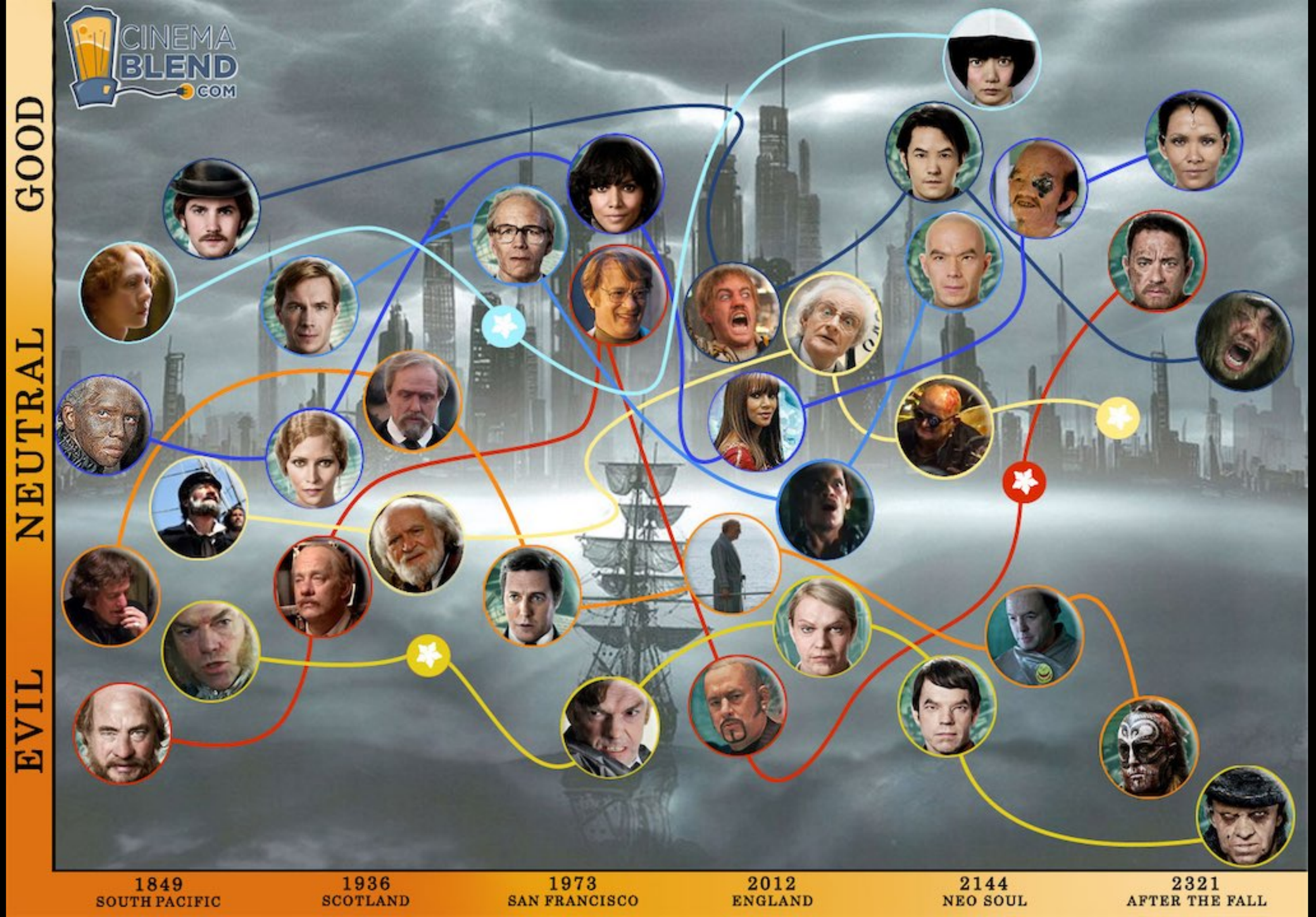
```
class TomHanksDamage
  attr_accessor :da_vinci_code, :damage

  def calc_damage()
    MovieModel.find_tom_hanks_movies().each do |movie|
      @damage = @damage + movie.damage_cost
      @da_vinci_code = true if movie.da_vinci?
    end
  end

  def damage_cost()
    if @da_vinci_code
      1_000_000_000_000
    else
      @damage
    end
  end
end
```

# Do this 👍

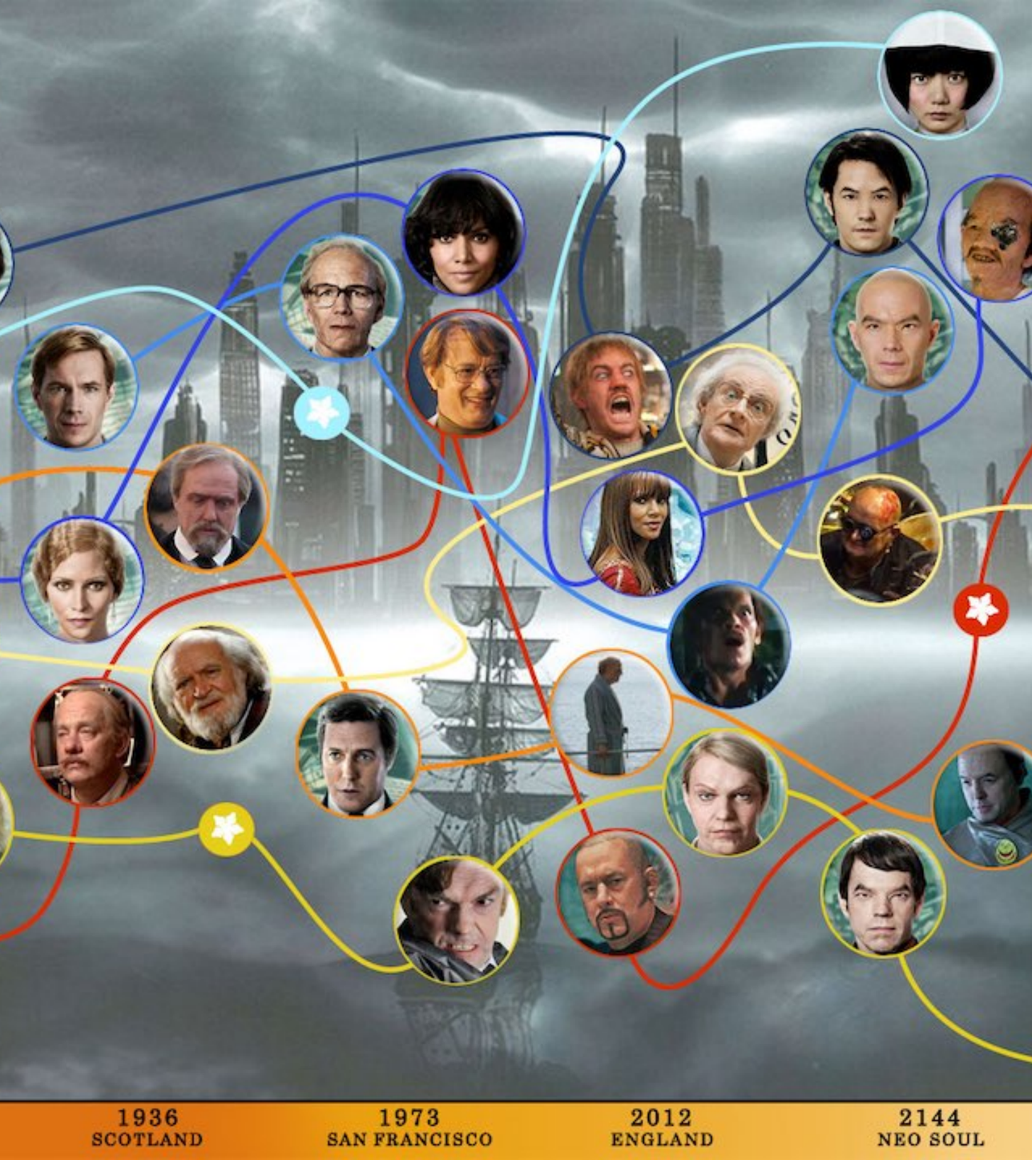
```
damage_cost = MovieModel.find_tom_hanks_movies().then{|movies|
  if movies.find(&:da_vinci?)
    1_000_000_000_000
  else
    movies.sum(&:damage_cost)
  end
}
```


CINEMA BLEND .COM
GOOD
NEUTRAL
EVIL
1849
SOUTH PACIFIC
1936
SCOTLAND
1973
SAN FRANCISCO
2012
ENGLAND
2144
NEO SOUL
2321
AFTER THE FALL

*"An exploration of how ones soul's sole actions of an individual lives/souls impact one another in the past, present and future..."*

*"An exploration of how ones soul's sole actions of an individual lives/souls impact one another in the past, present and future..."*

**GOOD LUCK DEBUGGING THAT**




# Immutability

Only create data - Never change it.

# Don't do this 😱

```
class CloudAtlas::Soul
  def initialize()
    @lives = []
  end

  def explore_soul(life)
    @lives << life
    CloudAtlas.every_single_other_soul.each do |other|
      other.impact_past_present_and_future(self)
    end
    do_something_philosophical_with_all_that_mutable_state()
  end
end

tom = CloudAtlas::Soul.new
tom.explore_soul("Dr Henry Goose")
tom.explore_soul("Isaac Sachs")
# this goes on for 172 minutes...
```

# Do this 👍

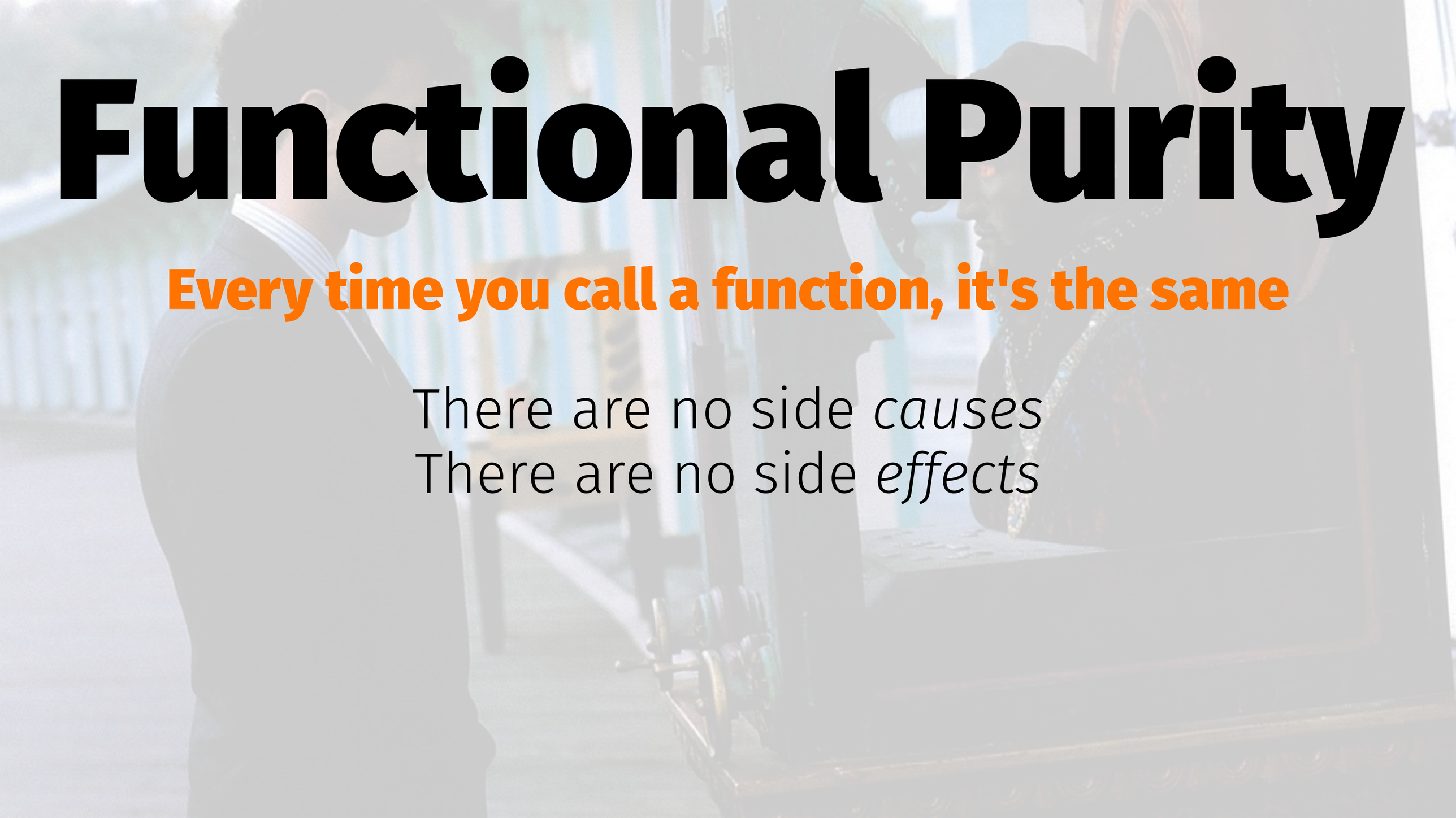
```
class CloudAtlas::Soul
  def initialize(lives=[])
    @lives = lives
  end

  def explore_soul(life, others)
    new_soul = CloudAtlas::Soul.new(@lives + [life])
    new_others = others.map{|other|
      other.immutably_transform_past_present_and_future(new_soul)
    }
    exploration = new_soul.do_something_philosophical(new_others)
    [new_soul, new_others, exploration]
  end
end
```

# Do this 👍

```
initial_state = [
  CloudAtlas::Soul.new,
  CloudAtlas.every_single_other_soul,
  []
]

["Dr Henry Goose", "Isaac Sachs"]
  .reduce(initial_state){|state, life|
    soul, others, explorations = state
    new_soul, new_others, exploration = soul.explore_soul(life, others)
    [new_soul, new_others, explorations + [exploration]]
  }
  # this still goes on for like 3 hours...
```



# Functional Purity

**Every time you call a function, it's the same**

There are no side *causes*
There are no side *effects*

# Functional Purity

All you have are arguments you get given
All you can do is compute something, and return it

**If you call it again with the same arguments,
it *must* return the same result**

# Functional Purity

→ **No** IO

→ **No** random

→ **No** current time

→ **No** threads

→ **No** state in other objects

→ **No** Tom Hanks gifs

# Don't do this 😱

```
def gif_my_tom(tom)
  tom_gif = TomHanksMemeService.download_a_tom_gif!(tom)
  cache_image(tom_gif)
  tom.image = tom_gif
  tom.updated_at = Time.now
  tom.save_to_the_database!
end
```

# Do this 👍

```
def actually_thats_a_whole_separate_talk()
  # I've only got 30 minutes 🤷‍♂️
  # short: Google "Functional Core, Imperative Shell"
  # longer: grab me for ☕/🍺 later
  # much longer:
  #  "Monads are just monoids in the category of endofunctors"
end
```

# So?... What actually *is* functional programming?

→ Statelessness

→ Immutability

→ Functional Purity

# What actually *is* functional programming? TL;DR?

## Expressing your logic like maths operations

**Stop** thinking of a list of imperative instructions
**Start** thinking how you can transform your data

Don't think about *how* it needs to happen

Think about *what* needs to happen

# But why?

## Seems like a lot of messing around...

# Functions fit in our brains

# Less to think about

# Less to go wrong

# The CPU couldn't care less

even if all your code was reams of global variables
and a mess of spaghetti `goto` statements

# Our brains need simple structures

→ We can create it

→ We can reason about it

→ We can test it

→ We can maintain it

→ We can change it

# But Ruby is an OO language though, right?

# Everything is a function if you squint hard enough

# Plain Old (Immutable) Ruby Objects

```ruby
class TomHanks
  def initialize(movies = [], damage_cost = 0)
    @movies = movies
    @damage_cost = damage_cost
  end

  def add_movie_damage(movie, damage)
    TomHanks.new(@movies + [movie], @damage_cost + damage)
  end
end

tom = TomHanks.new()
destructive_tom = tom.add_movie_damage("Forest Gump", 50_000_000)
```

# Plain Old Ruby Objects

I thought you said **state** was bad, and we need to do statelessness?

# Plain Old Ruby Objects

I thought you said **state** was bad, and we need to do statelessness?

**Everything is a function if you squint hard enough**

# Everything is a function if you squint hard enough

`self` is just an implicit variable passed to a function.

If you treat it like that, it's *ok*.

# What would Python do?

```
class TomHanks:
  def __init__(self, movies = [], damage_cost = 0):
    self.movies = movies
    self.damage_cost = damage_cost

  def add_movie_damage(self, movie, damage):
    return TomHanks(self.movies + [movie], self.damage_cost + damage)

tom = TomHanks()

destructive_tom = tom.add_movie_damage("Forest Gump", 50000000)

also_destructive_tom = TomHanks.add_movie_damage(tom, "Forest Gump", 50000000)
```

# What would Elixir do?

```elixir
defmodule TomHanks do
  defstruct [movies: [], damage_cost: 0]

  def add_movie_damage(tom, movie, damage) do
    %TomHanks{tom |
      movies: [tom.movies | movie],
      damage_cost: tom.damage_cost + damage
    }
  end
end

tom = %TomHanks{}
destructive_tom = TomHanks.add_movie_damage(tom, "Forest Gump", 50_000_000)
```

# We get something else for free with Ruby methods

# Blocks

# Blocks

## Let you pass an anonymous function to any Ruby method

```
class Apollo13
  # ...
  def stir_oxygen_tanks?
    @oxygen_tanks.any?{|tank| yield tank}
  end
end
```

# Blocks let you choose different implementations

```
class Apollo13
  # ...
  def stir_oxygen_tanks?
    @oxygen_tanks.any?{|tank| yield tank}
  end
end

apollo13.stir_oxygen_tanks?{|tank|
  !tank.stirred_today?
}

apollo13.stir_oxygen_tanks?{|tank|
  tank.pressure_sensor_malfunctioning?
}
```

# Two ways to call blocks

## { } vs do ... end

```
apollo13.stir_oxygen_tanks?{|tank| tank.stirred_today? }

apollo13.stir_oxygen_tanks? do |tank|
  tank.stirred_today?
end
```

# Controversial block syntax opinion

Use `{}` for when you care about the returned value

Use `do ... end` for side effects

***Ignore line count***

# Controversial block syntax opinion

```
should_stir = apollo13.stir_oxygen_tanks?{|tank|
  tank.pressure_sensor_malfunctioning?
}

apollo13.stir_oxygen_tanks do |tank|
  if tank.number == 2
    tank.explode!
    tom.say "Houston, we have a problem"
  end
end
```

**Communicate intent with** { } **or** do **...** end

**It's not about dogmatic whitespace rules**

# If you use Rubocop

## Set block delimiters to `semantic`

```
#.rubocop.yml
Style/BlockDelimters:
  EnforcedStyle: semantic
```

# `Object#then` **is super cool**

remove state from your context

put it in a block

```
damage_cost = MovieModel.find_tom_hanks_movies().then{|movies|
  # I can see `movies` here
  do_stuff(movies)
  do_other_stuff(movies)
}
# I can not see `movies` here
# statelessness is preserved
```

# `then` **is super cool**

The implementation is trivial and elegant though

```
class Object
  def yield_self
    yield self
  end

  alias then yield_self
end
```

# The thing is though...

# Blocks don't really exist

Not as objects at least.
Syntax sugar for passing a bunch of code

# What if we want to grab hold of a block?

```
class Apollo13
  # ...
  # prefix last argument with &
  def stir_oxygen_tanks?(&should_stir)
    # then you can use it like any other object
    @oxygen_tanks.any?{|tank| should_stir.(tank)}
  end
end
```

# Now we have it, we can pass it around

```
class Apollo13
  # ...
  def stir_oxygen_tanks?(&should_stir)
    # prefix it with & again to pass as a block
    @oxygen_tanks.any?(&should_stir)
  end
end
```

# What is this captured block thing?

```ruby
def do_block(&block)
  block.inspect
end

do_block { "I'm a 📦" }
# "#<Proc:0x00007f8e35173c00@(irb):35>"
```

# A block you grab with `&` is a `Proc` object

# procs =~ lambdas (mostly...)

We'll use them interchangably in this talk

There are some subtle differences

We'll ignore that for today

# Procs (and lambdas) are first class functions

"first class" just means something you can assign to a value, and pass around like anything else

# procs are first class anonymous functions

## (aka lambdas)

```
my_proc = proc{|x,y| x + y}
my_proc.(1,2)
# 3

my_lambda = lambda{|x,y| x + y}
my_lambda.(1,2)
# 3

my_lambda = ->(x,y){x + y}
my_lambda.(1,2)
# 3
```

# Lambdas can be passed as blocks with the & operator

```
missions = [apollo11, apollo12, apollo13, ...]

mission_result = ->(mission){
  if mission.tom_hanks_is_commander?
    "Houston, we have a problem"

  else
    "success"
  end
}

missions.map(&mission_result)
["success", "success", "Houston, we have a problem", ...]
```

# **Lambdas can be passed as blocks with the** & **operator**

More about & later...

# Currying

```
curried_add =
  ->(x){
    ->(y){
      ->(z){ x + y + z}
    }
  }

curried_add.(2)           #<Proc:0x0000125...>
curried_add.(2).(3)       #<Proc:0x0000126...>
curried_add.(2).(3).(4) # 9
```

# `curry` creates a curried lamba

```
add = ->(x,y,z){x + y + z}   #<Proc:0x0000123...>
add.(2,3,4)                  # 9

curried_add = add.curry      #<Proc:0x0000124...>
curried_add.(2)              #<Proc:0x0000125...>
curried_add.(2).(3)          #<Proc:0x0000126...>
curried_add.(2).(3).(4)      # 9
```

# Lambdas can be partially applied

Kinda like dynamically setting default args

Useful if you need to pass extra args and do "dependency injection"

(like config, or some other context)

```
add_two = curried_add.(2)    #<Proc:0x0000127...>
add_two.(3)                  #<Proc:0x0000128...>
add_two.(3).(4)              # 9
add_two.(3, 4)               # 9
```

# Lambdas can be partially applied

```
mission_result -> (flight_director, crew){
  if flight_director.is_ed_harris?
    "Failure is not an option"
  else
    successful_if_crew_does_ok(crew)
  end
}

# but our report function doesn't know about flight directors...
def mission_report(mission, &result_lambda)
  "for #{mission.name}, the result was: #{result_lamba.call(mission.crew)}"
end

# so we curry our lambda
mission_result_with_ed = mission_result.curry.call(ed_harris)
mission_report(apollo13, &mission_result_with_ed)
# Failure is not an option
```

# Lambdas can be composed with << and >>

```
add_two = ->(x) {x + 2}
times_three = ->(x) { x * 3}

(add_two << times_three).(4)
# add_two.(times_three.(4))
# (4 * 3) + 2 == 14

(add_two >> times_three).(4)
# times_three.(add_two.(4))
# (4 + 2) * 3 == 18
```

# There are many ways to call a lambda

```
my_lambda.(foo)
my_lambda.call(foo)
my_lambda[foo]
my_lambda === foo # so you can use in case statements
my_lambda.yield(foo)
```

`yield`... That's interesting...

# Remember blocks?

```
class Apollo13
  # ...
  def stir_oxygen_tanks?
    @oxygen_tanks.any?{|tank| yield tank}
  end
end

apollo13.stir_oxygen_tanks?{|tank|
  !tank.stirred_today?
}
```

# Remember blocks?

Blocks are called with `yield`...
lambdas are called with `yield`

This gives us insight into how `yield` works

# Remember blocks?

So these code snippets are equivalent

```
def stir_oxygen_tanks?
  @oxygen_tanks.any?{|tank| yield tank}
end

def stir_oxygen_tanks?(&should_stir)
  @oxygen_tanks.any?{|tank| should_stir.yield(tank) }
end
```

# `yield` is not doing anything magic

Just calling the implicit block

# What was that thing about the & operator before?

# & converts an object into a block

...

# What was that thing about the & operator before?

# & converts an object into a block

Objects... like lambdas and procs

# & converts an object into a block

Which is how we can pass a lambda as a block

```
missions = [apollo11, apollo12, apollo13, ...]

mission_result = ->(mission){
  if mission.tom_hanks_is_commander?
    "Houston, we have a problem"
  else
    "success"
  end
}

missions.map(&mission_result)
["success", "success", "Houston, we have a problem", ...]
```

# & converts an object into a block

It works with other things too. Like symbols...

```
[1, -3, 2, -4].select(&:positive?)
# [1,2]
```

# Under the hood, it's just calling `to_proc` on the symbol

```ruby
class Symbol
  def to_proc # simplified...
    ->(obj){ obj.send(self) }
  end
end

[1, -3, 2, -4].select(&:positive?)

is_pos = :positive?.to_proc
[1, -3, 2, -4].select(&is_pos) # [1, 2]

is_pos2 = ->(x){x.send(:positive?)
[1, -3, 2, -4].select(&is_pos2)
```

# You can convert *your* code to lambdas too

Use `&` operator, and implement `to_proc`

```
class Adder
  def initialize(addend)
    @addend = addend
  end

  def to_proc
    ->(x){ x + @addend }
  end
end

add_two = Adder.new(2)
[2,5,1,7].map(&add_two)
# [4, 5, 3, 9]
```

# Functions love recursion

```
def factorial(n, acc=1)
  if n <= 1
    acc
  else
    factorial(n-1, n*acc)
  end
end

factorial(1) # 1
factorial(5) # 120
```

# Functions love recursion

## Ruby's stack *doesn't* love recursion though...

```ruby
factorial(100000) # Oh oh... 🤯
# stack level too deep (SystemStackError)
```

```
Traceback (most recent call last):
        10080: from factorial.rb:14:in `<main>'
        10079: from factorial.rb:5:in `factorial'
        10078: from factorial.rb:5:in `factorial'
        10077: from factorial.rb:5:in `factorial'
        10076: from factorial.rb:5:in `factorial'
        10075: from factorial.rb:5:in `factorial'
        10074: from factorial.rb:5:in `factorial'
        10073: from factorial.rb:5:in `factorial'
         ... 10068 levels...
            4: from factorial.rb:5:in `factorial'
            3: from factorial.rb:5:in `factorial'
            2: from factorial.rb:5:in `factorial'
            1: from factorial.rb:5:in `factorial'
factorial.rb:5:in `factorial': stack level too deep (SystemStackError)
```

# Each function call goes on the stack

Too many, and you'll overflow it eventually

# Luckily, we have a tail call optimisation

A function is "tail recursive" if the *last* thing a function does is return a value and **nothing** else afterwards

```
def factorial(n, acc=1)
  if n <= 1
    # last thing in this branch ✅
    acc
  else
    # last thing in this branch ✅
    factorial(n-1, n*acc)
  end
end
```

# Enabling tail call optimisation

```ruby
# main.rb
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
require_relative 'factorial'

factorial(100000)
# 2824229407960347874293421 5...
# Nice 👍
```

# What did all those cool tricks have in common?

## The Three things

→ statelessness

→ immutability

→ functional purity

→ (And Tom Hanks trashing stuff)

# Statelessness, immutability, and purity

If you take anything away from today:

# Statelessness, immutability, and purity

If you take anything away from today:

**90% of functional programming is about those three things**

They let you make functions that fit in your brain

AND THAT'S ALL I HAVE

TO SAY ABOUT THAT.

# Thank you!

## Cool Functional Tricks In Ruby

Julian Doherty
@madlep
juliandoherty.com