A Journey Through Ruby Concurrency

RubyConf Brasil 2016

@jerrydantonio

I'd like to start by saying Thank You to Locaweb.
Locaweb sponsored my travel visa, assisted me with all my travel arrangements, and helped me through the visa process.
They are tremendous people who have gone out of their way to assist me with this visit.
It's fair to say that without Locaweb I would not be here.
So please give a big round of applause to Locaweb!

Jerry D'Antonio

Akron, Ohio, USA
Software Developer
Test Double

@jerrydantonio
github.com/jdantonio
concurrent-ruby.com

My name is Jerry D'Antonio
I am from Akron, Ohio, USA
I am a Software Developer at Test Double, a consulting agency based in Columbus, Ohio.
I am @jerrydantonio on Twitter
And jdantonio at GitHub

Most importantly, with respect to this presentation, I created Concurrent Ruby.
The Concurrent Ruby gem is a suite of concurrency tools used by many well-known projects including Rails, Sidekiq, Sucker Punch, and Microsoft's Azure tools for Ruby.

You may have heard of Akron, Ohio.
There's a local kid who you may have heard of.
He's a pretty good basketball player.
His name is LeBron James.
So I am definitely NOT the most famous person from Akron, Ohio.
Not even close. :-)

But we aren’t here to talk about basketball. We’re here to talk about Ruby concurrency.

A few weeks ago Koichi Sasada, aka ko1, proposed a new concurrency model for Ruby 3.

And now everyone is talking about concurrency.

Even more than usual.

Including me.

Especially me.

Which is good. It's an important topic, and critical to Ruby's future.

But we need to have the right conversations.

When discussing Ruby concurrency I frequently hear people say wrong things.

People say things like:
“Ruby should do what Erlang does.”
Or “Ruby should do what Node.js does.”
Or “Ruby should do what Go does.”
Or “Ruby should do what… some other language does.”

Although well meaning, comments like this can sometimes be unhelpful.

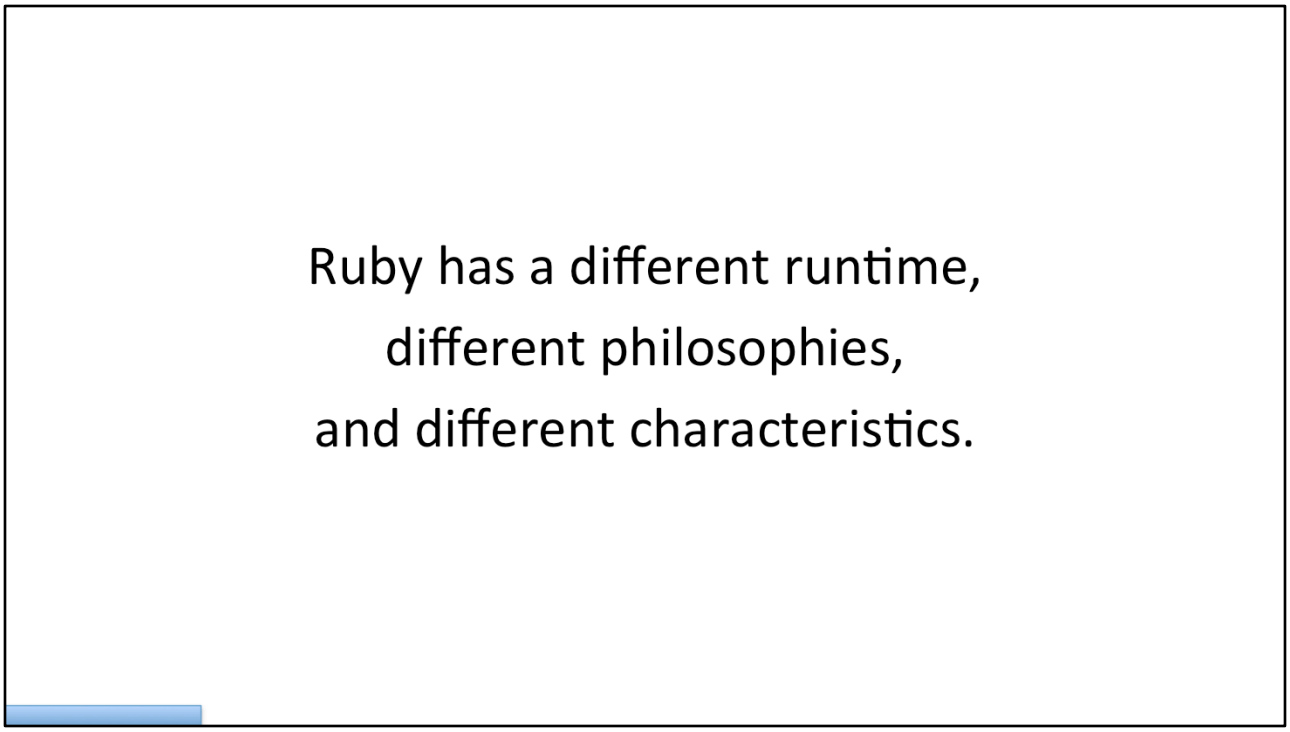
Ruby can’t necessarily do what Erlang does, or what Node does, or what language X does.

Nor should it.

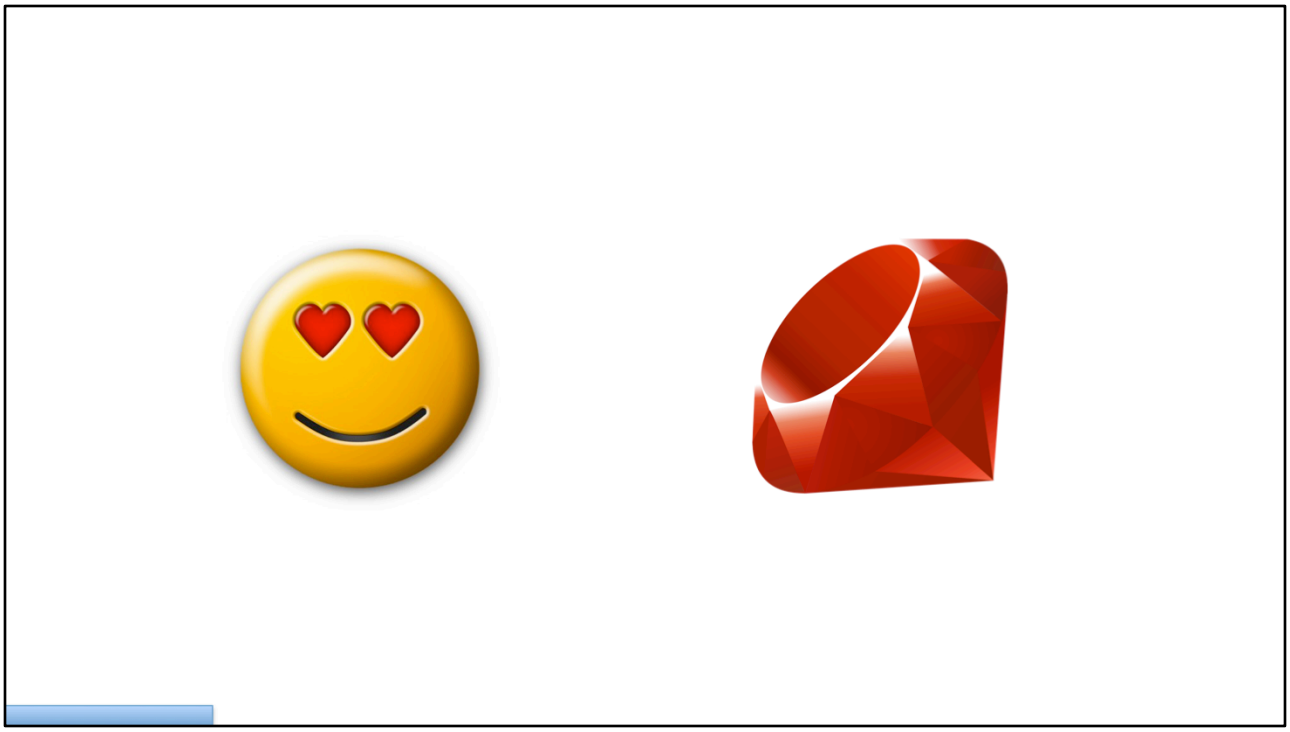
Because Ruby isn't Erlang or JavaScript or any other language.

Ruby is Ruby 

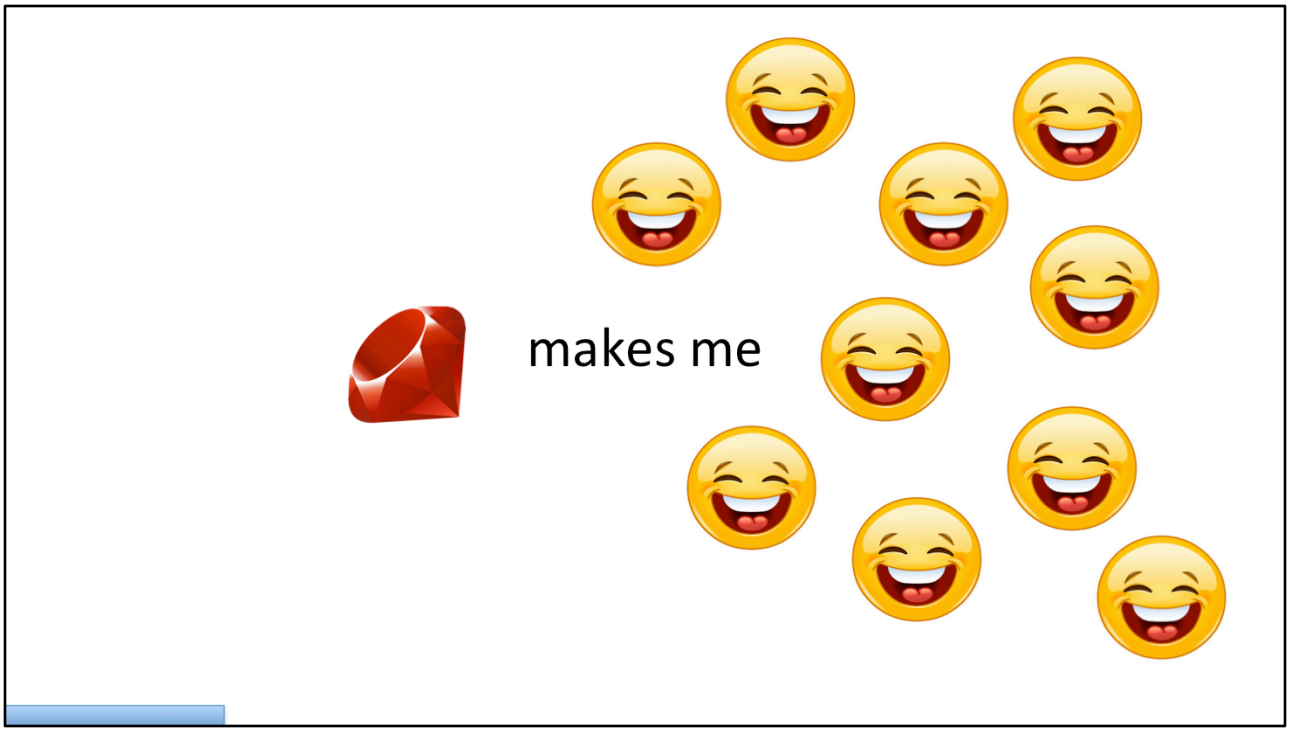
Ruby is designed to solve different problems.



Ruby isn't being used to build telephony systems.
Ruby isn't being used to build large-scale chat systems.
Ruby isn't being used for systems programming.
And that's OK.
Ruby solves some problems very well, and those are the types of applications Ruby is used for.



I love Ruby.

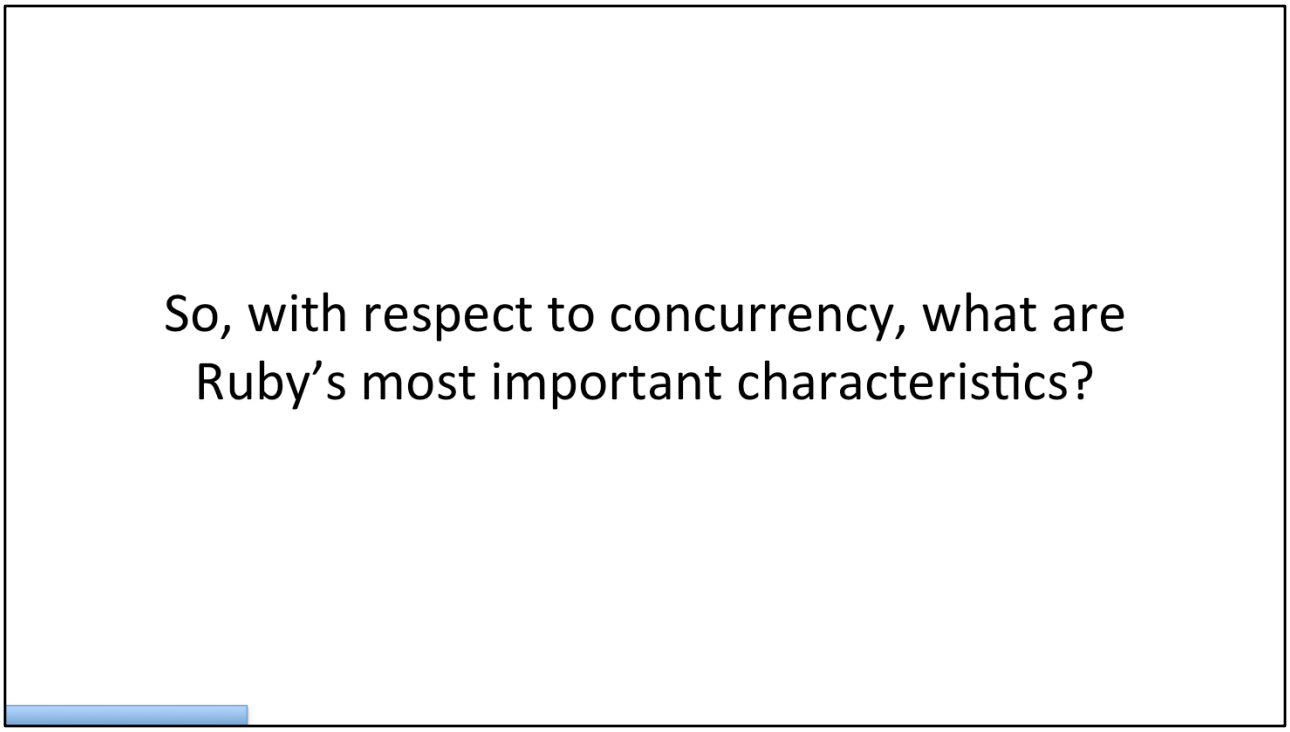


Ruby makes me happy.
Ruby makes many, many developers happy.
We know this because there are 1300 people here, today, who love Ruby.

Although Ruby can—and should—take inspiration from other languages.

Ruby must embrace Ruby.

Ruby must embrace the things about Ruby that we love.
Ruby must embrace the things about Ruby that make us happy.



What are the things that we, Ruby developers, must consider when writing concurrent applications?
What are the things that we, the Concurrent Ruby team, must think about when designing concurrency tools?
What are the things the Ruby core team needed to think about when designing the concurrency model for Ruby 3?

Ruby is…

Shared memory

Reference-based

Object oriented

Everything in Ruby is an object.

Ruby has no value types, no true primitives.

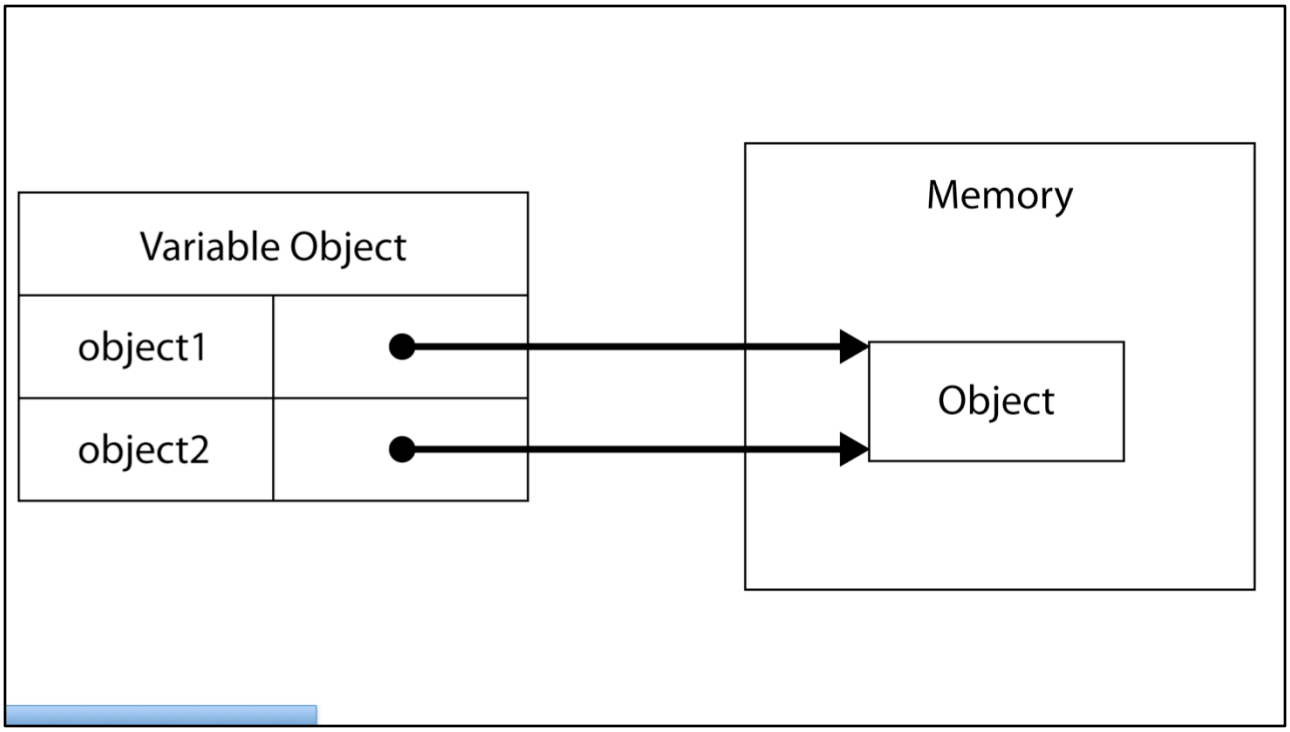
Even booleans, symbols, and integers are objects.

As most of you know…

Variables are not values.

Variables are references to values.

References point to memory locations on the heap.



Strictly speaking, Ruby is a *call by sharing* language.
Pass an object reference to a method and the reference gets copied.
The object does not.

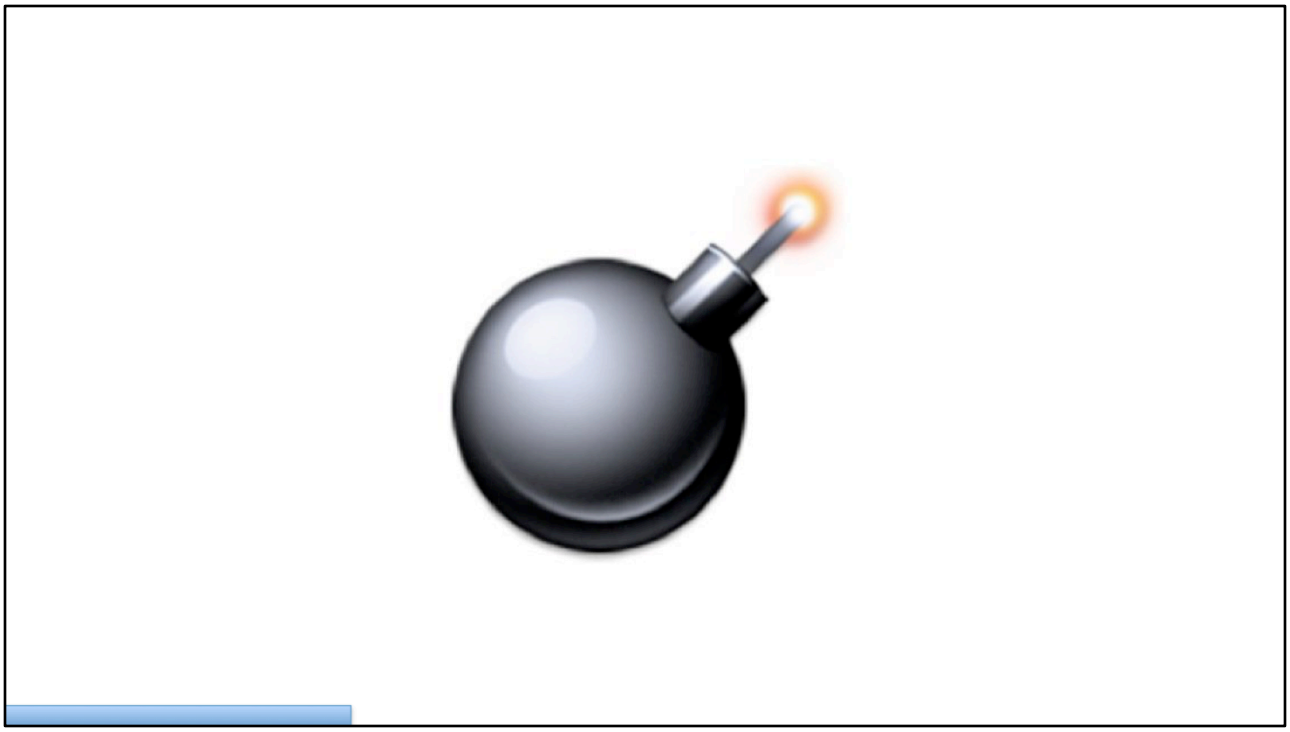
Objects can be modified from within a method by following the reference.

And this behavior exists when we pass variables across thread boundaries.

Strictly speaking, Ruby is a *call by sharing* language.
Pass an object reference to a method and the reference gets copied.
The object does not.

So two or more threads can each have references to the same object, located in the same memory on the heap.

Which means that, in theory, two threads on two processor cores can attempt to simultaneously modify the same memory.



And when we do that, bad things happen.

Not on MRI, of course, because it only uses one core and it has a global interpreter lock (GIL).

But that's sort of a problem, too.

This model provides safety when writing concurrent applications, but it has serious implications.
I'm not going to talk about this because there are plenty of blog posts and conference talks about the global interpreter lock.
Including my talk from last year's RubyConf USA, called "Everything You Know About the GIL is Wrong."

Shared memory, reference-based, object oriented languages offer many advantages.

Which is why they are so popular.

But they aren’t great for concurrency.

Full parallelism with shared-memory, reference-based languages can be very fast and efficient, but is also very difficult to get right.

Locking is hard.

Shared memory, reference-based languages:

Ruby

Python

C, C++

Java

C#

Go

Shared memory, reference-based languages include:
…
And many others.
Basically, most of today's dominant programming languages.

Functional languages like Erlang, Clojure, and Haskell are generally considered better for concurrency.

But they are very different languages.

They favor values over references,
enforce immutability,
and in some cases offer memory isolation.

Erlang, for example...

Has only values, not references.

Has only immutable variables.

Has a very small set of value types.

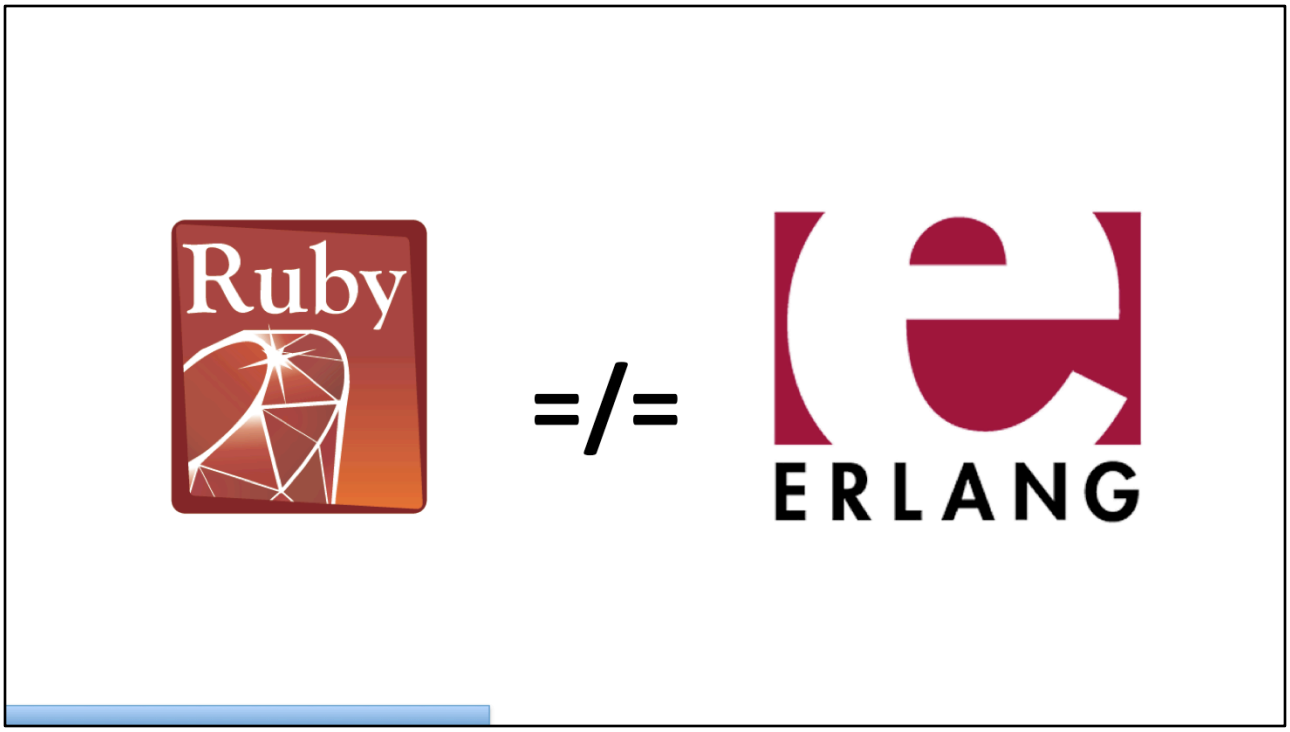
And enforces strict memory isolation.

This is all very good for concurrency.

But isn't necessarily all good.

Or necessarily better.

Ruby and Erlang are very different languages.

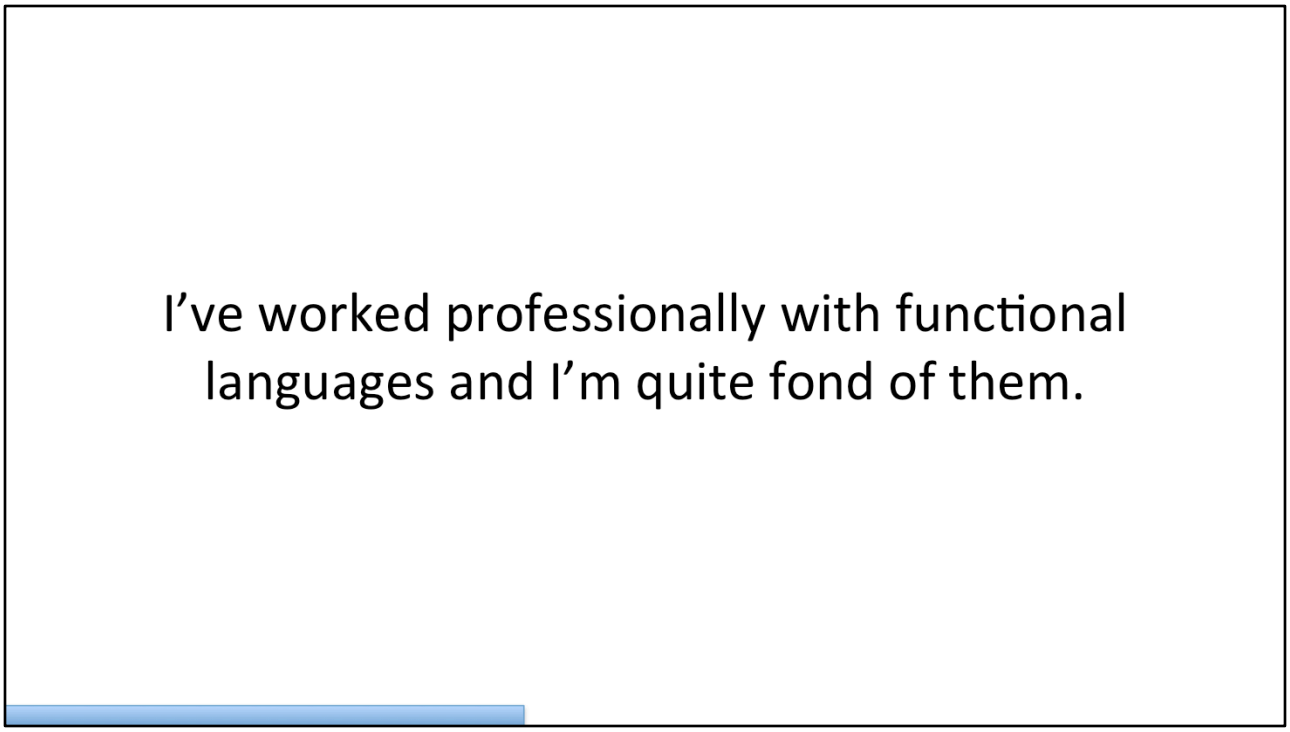


This one is for the Erlangers in the audience.

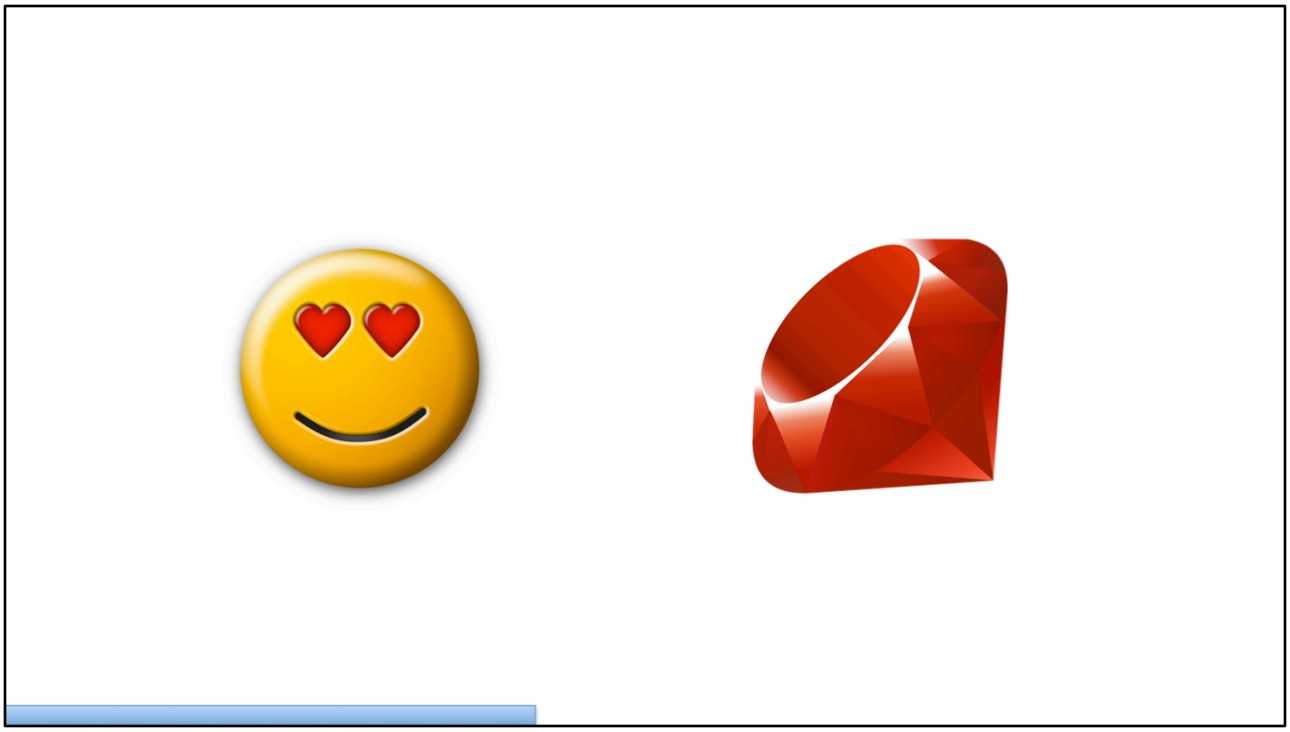
Functional languages like Erlang require different paradigms and patterns, but also pose different challenges.

That's OK. This isn’t necessarily good nor is it necessarily bad.
It’s just different.

Some people like programming in functional languages.



I even created a gem called functional-ruby where I experiment with other stuff.

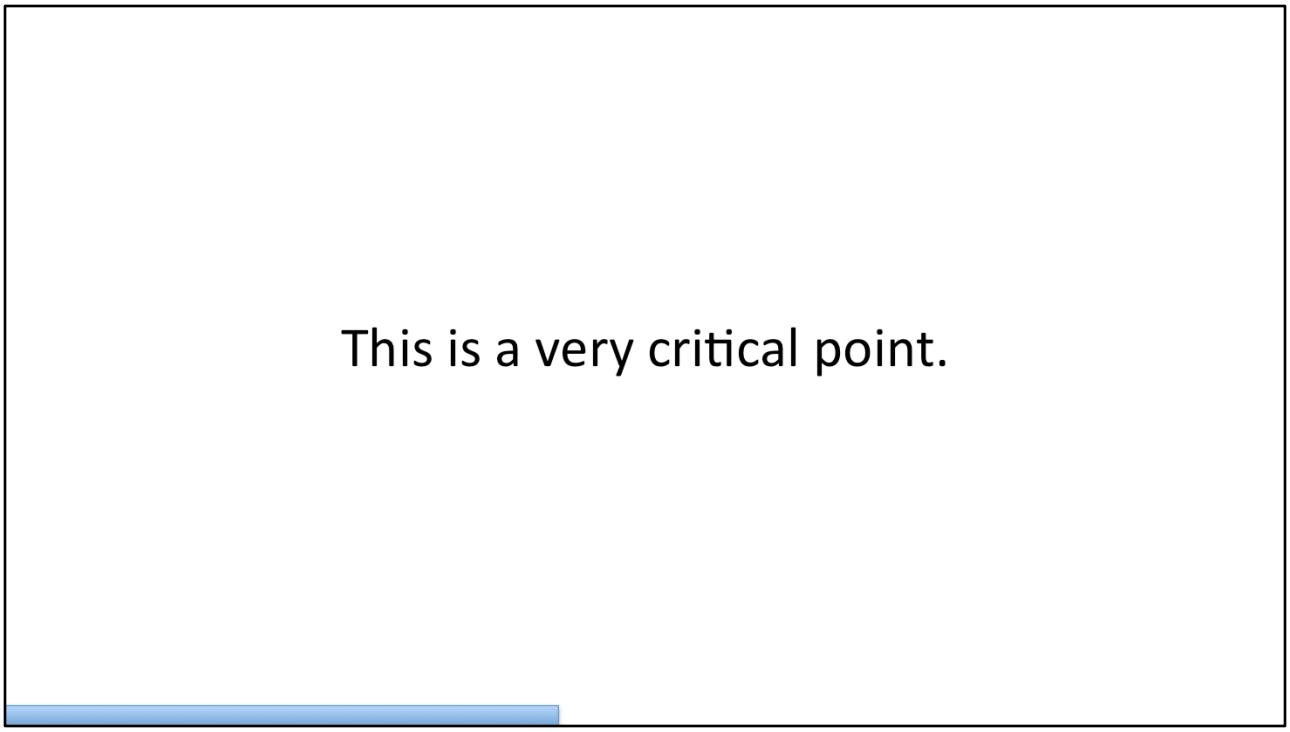


But I love Ruby.

And so do many, many programmers.

Ruby cannot have the same concurrency model as most functional languages without becoming a very different language.

Repeat this.



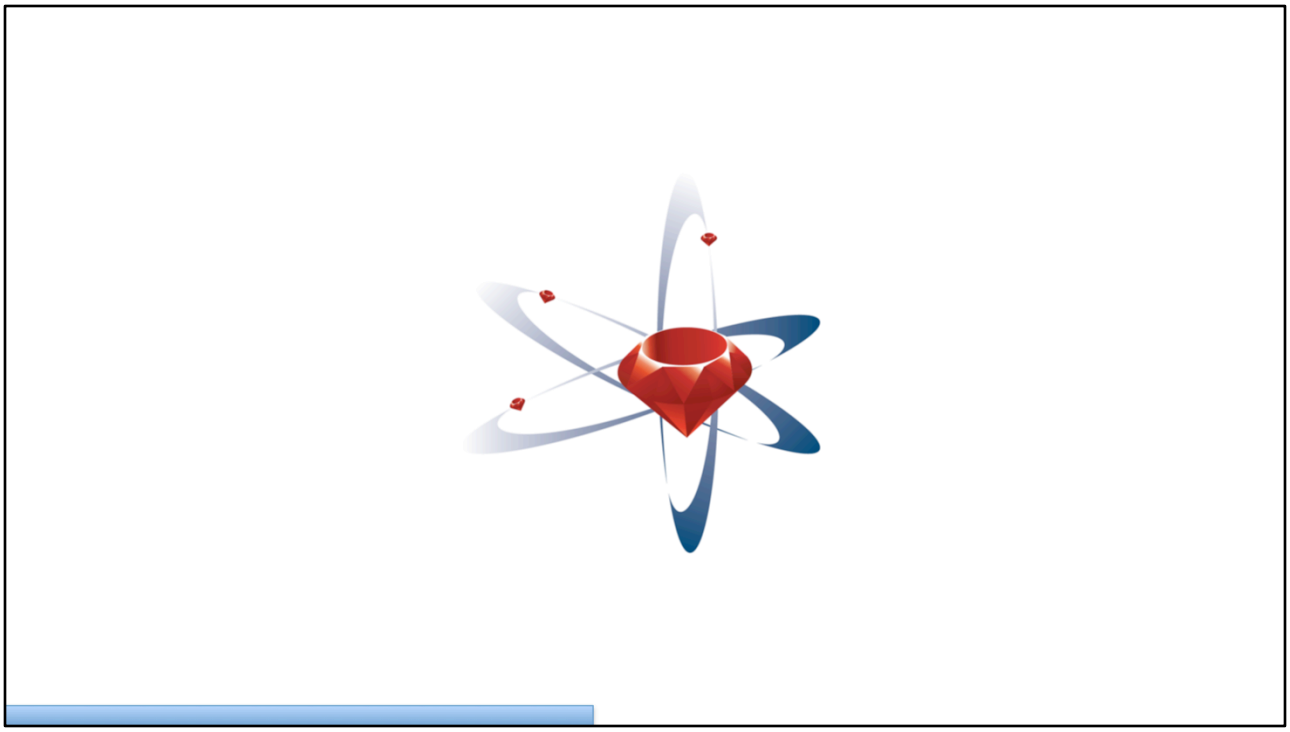
We want Ruby to keep being Ruby.

Ruby simply cannot copy what Erlang, Clojure, or Haskell does without ceasing to be Ruby.

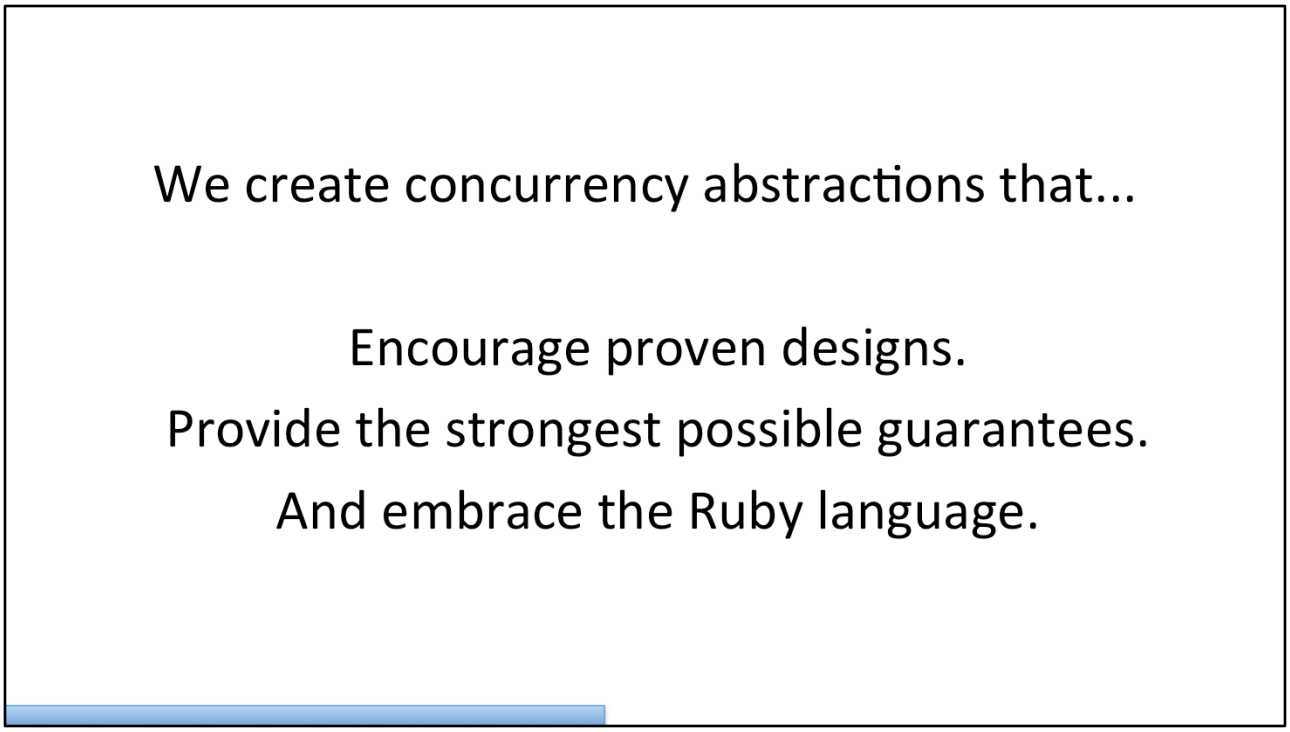
So what do we do?

And by “we” I mean the Concurrent Ruby team.

Because we’ve been working on this problem for several years.



So let’s talk about Concurrent Ruby.



Here's what we do:

So how do we do that?

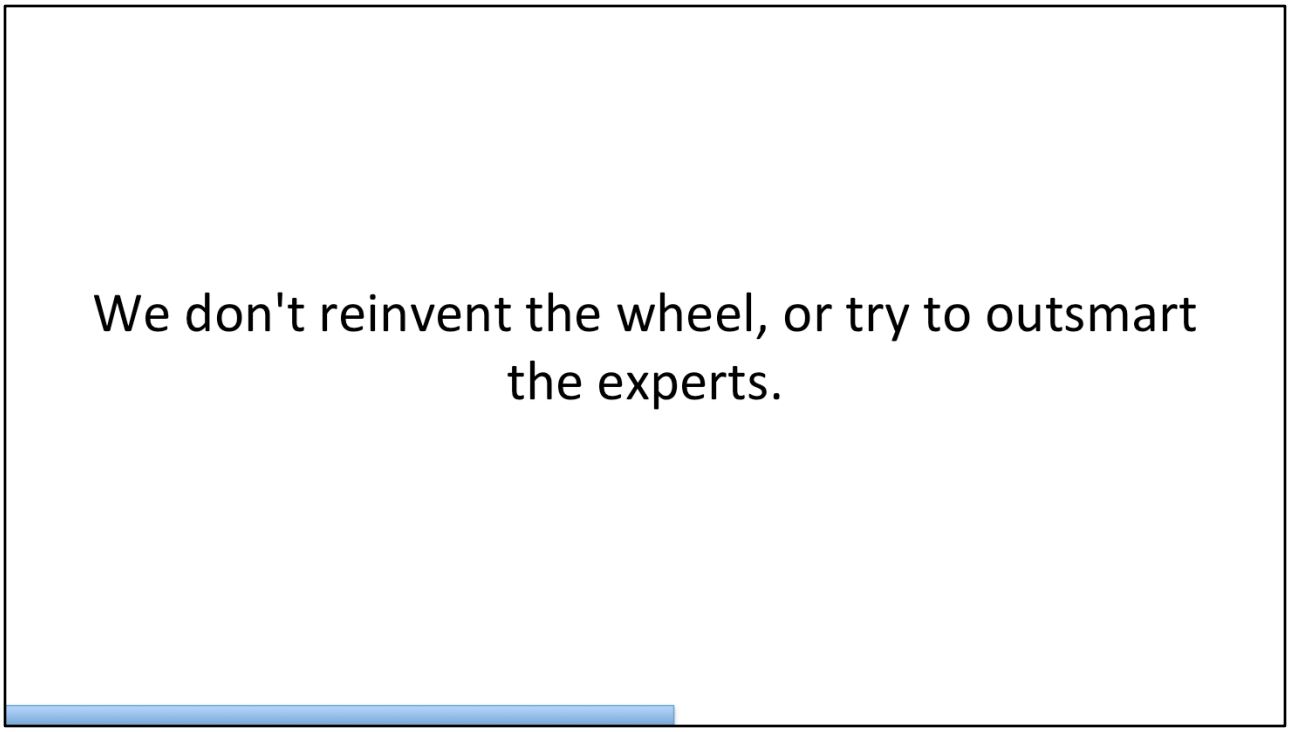
We do three things…


1

Concurrent Ruby takes its *inspiration* from other languages.

We leverage *ideas* from Erlang, Clojure, Java, JavaScript, and other languages.

As well as from academic research.



Even though several members of the team are recognized experts.

We create abstractions which enable Ruby programmers to easily implement those proven patterns and practices.

And when programmers follow the guidelines for what we provide, they create safe concurrent systems.


2

Concurrent Ruby provides the strongest possible thread safety guarantees.

Emphasis on the word “possible.”

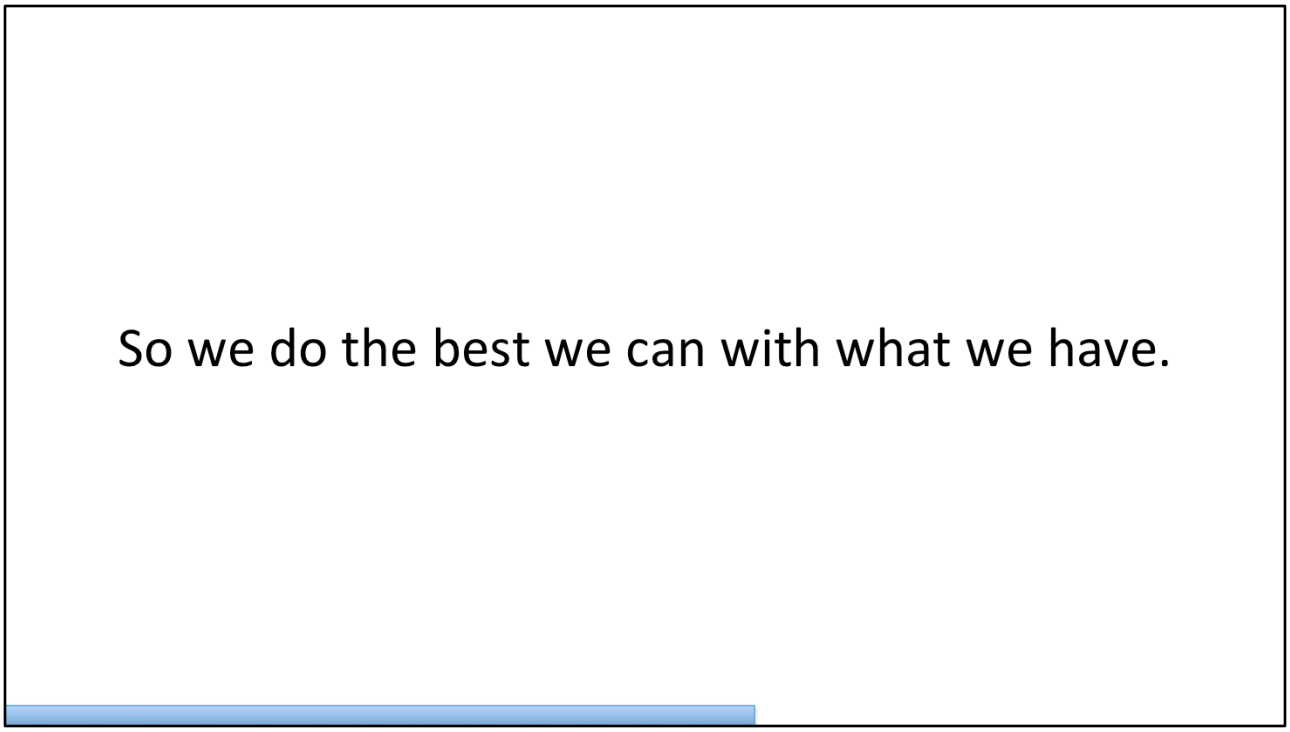
We can't provide stronger guarantees than what the runtime provides.

No concurrency library can.

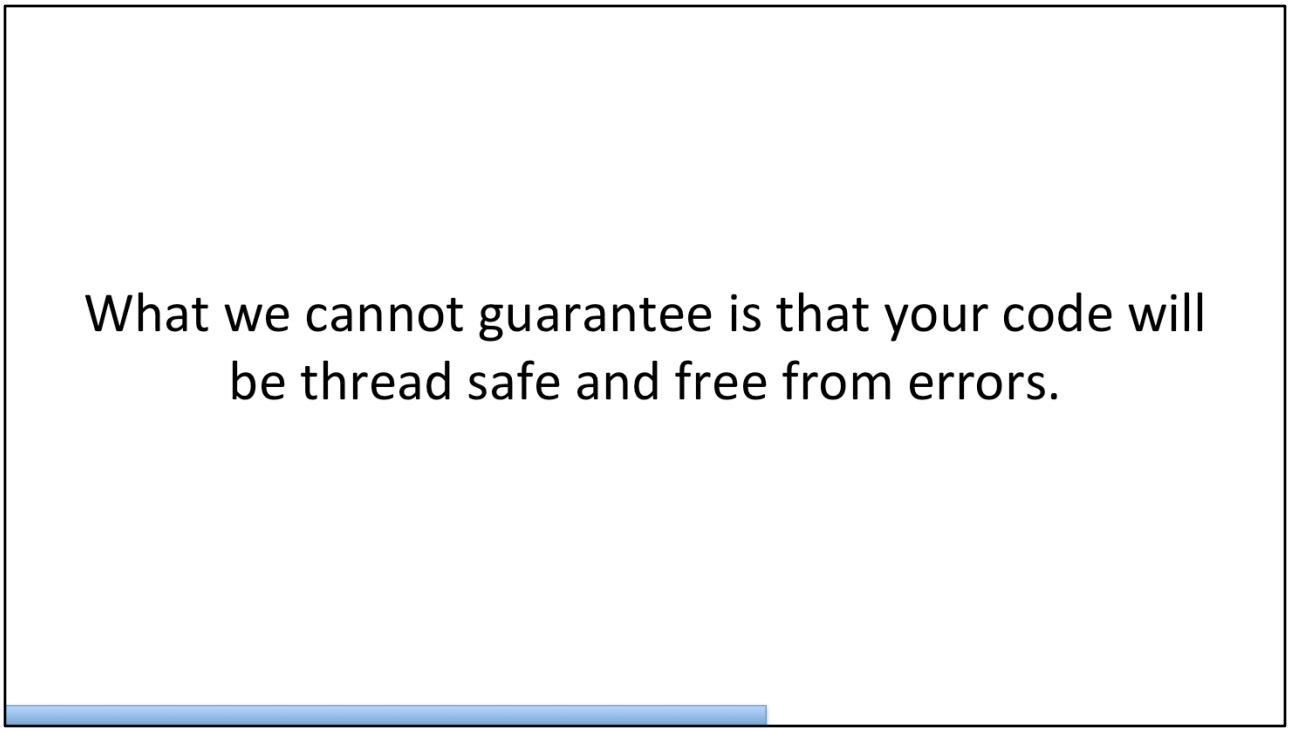
We use the runtime, we don't replace it.

Unfortunately, Ruby does not have a formal memory model.

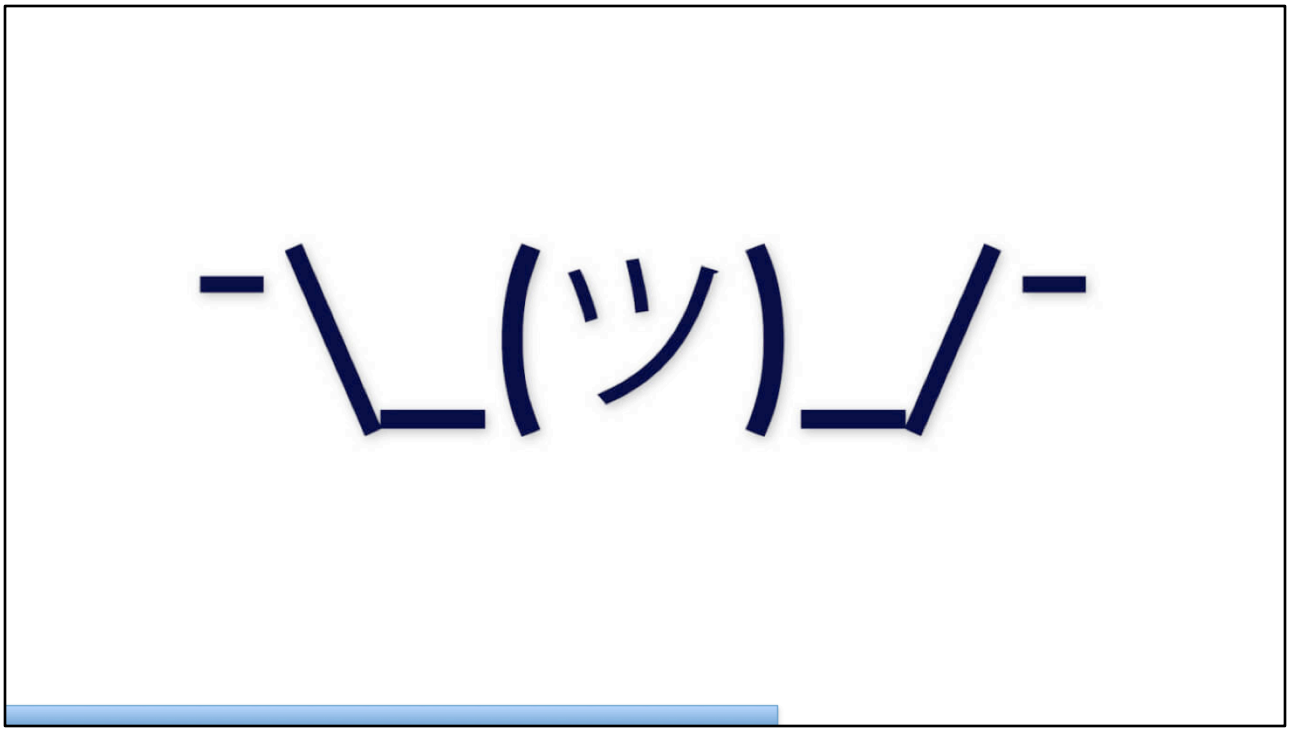
For those not familiar with the term memory model:
"A memory model describes the interactions of threads through memory and their shared use of the data." – Wikipedia
Basically, this is a formal definition of much of what we are talking about today.



We optimize our internals for each of the three major runtimes.
We guarantee that our abstractions are free from deadlocks, race conditions, and other concurrency errors.
And we guarantee that our abstractions are thread safe.



What we cannot guarantee is that your code will be thread safe and free from errors.
No Ruby concurrency library can.
Honestly, most languages can't.
There are many, many ways to make mistakes when creating concurrent systems.

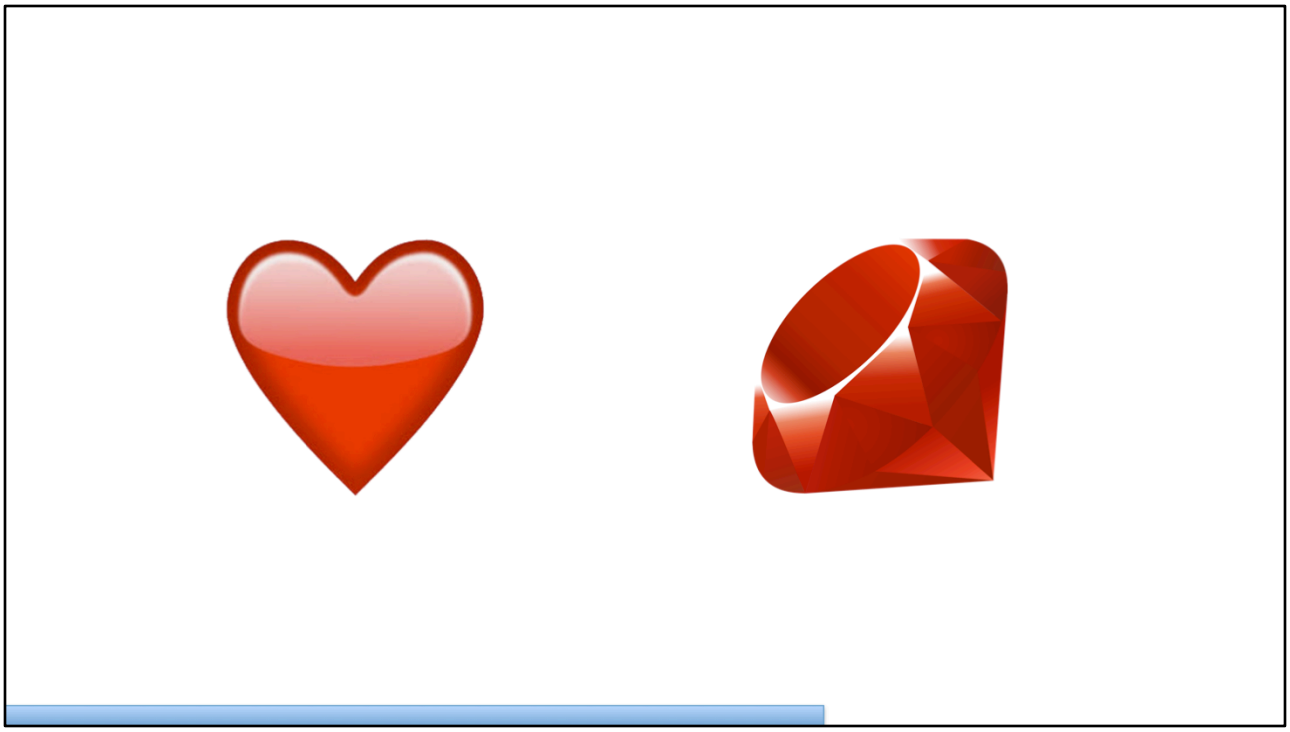


Concurrency is hard. We can make it easier, but we cannot necessarily make it easy.

We can only guarantee that if you follow the guidelines and use our tools the way they were intended that you'll probably be OK.

Which is the only guarantee any concurrency library can give you.

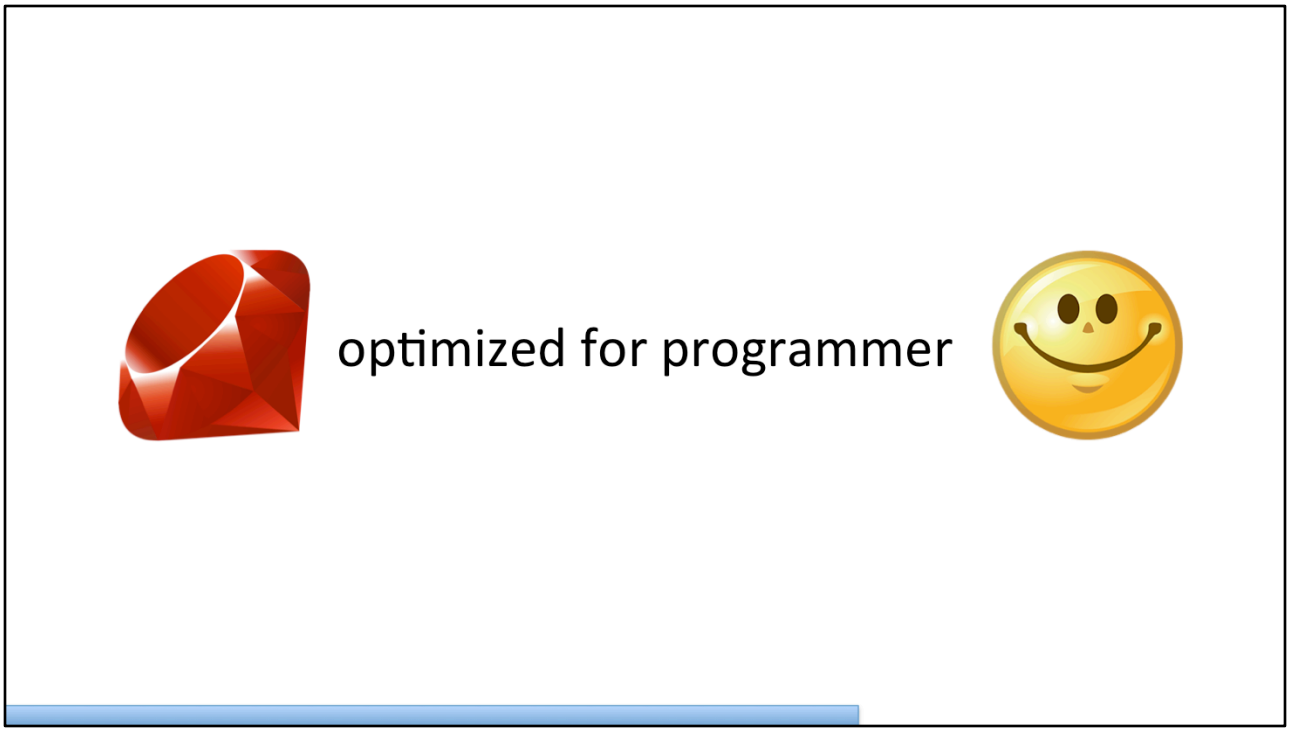
The last, and possibly most important thing we do, is to embrace the Ruby language.



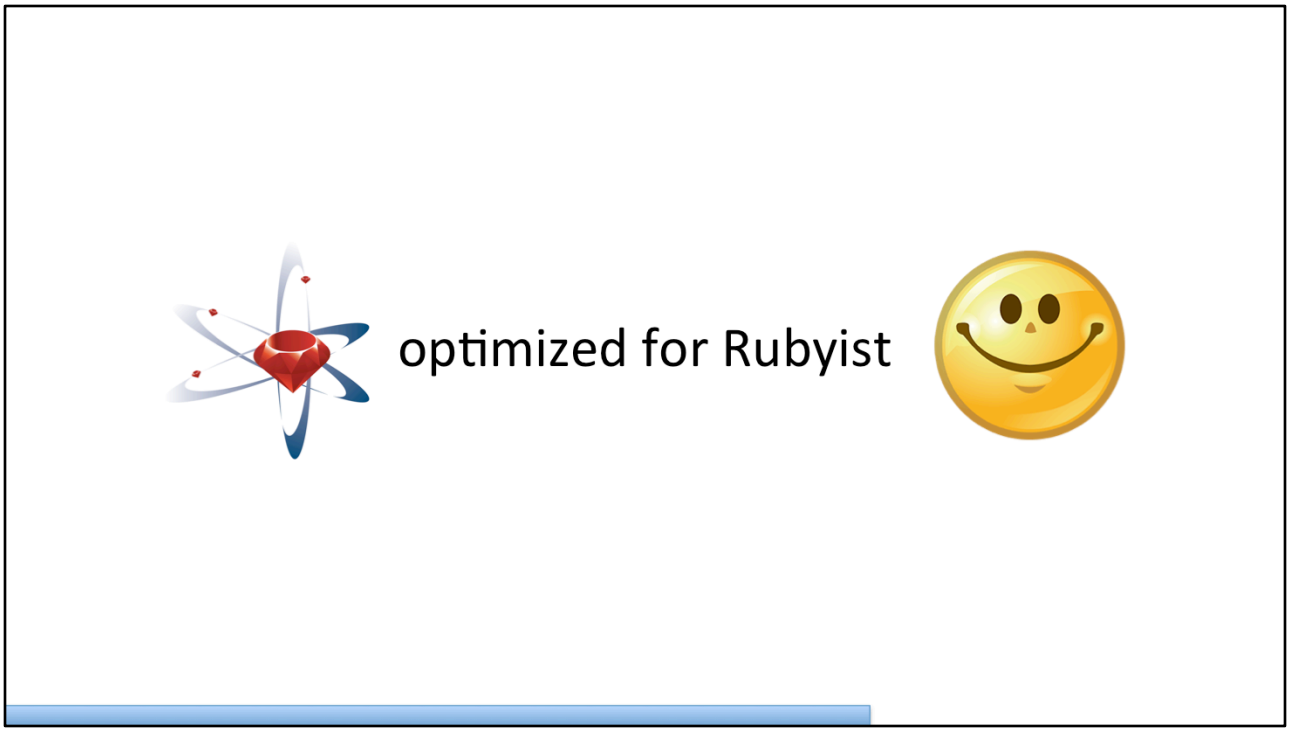
We love Ruby. So we put this love into everything we build.

Even though we take inspiration from other languages, we don't directly copy them.

Instead, we interpret them in a uniquely Ruby way.

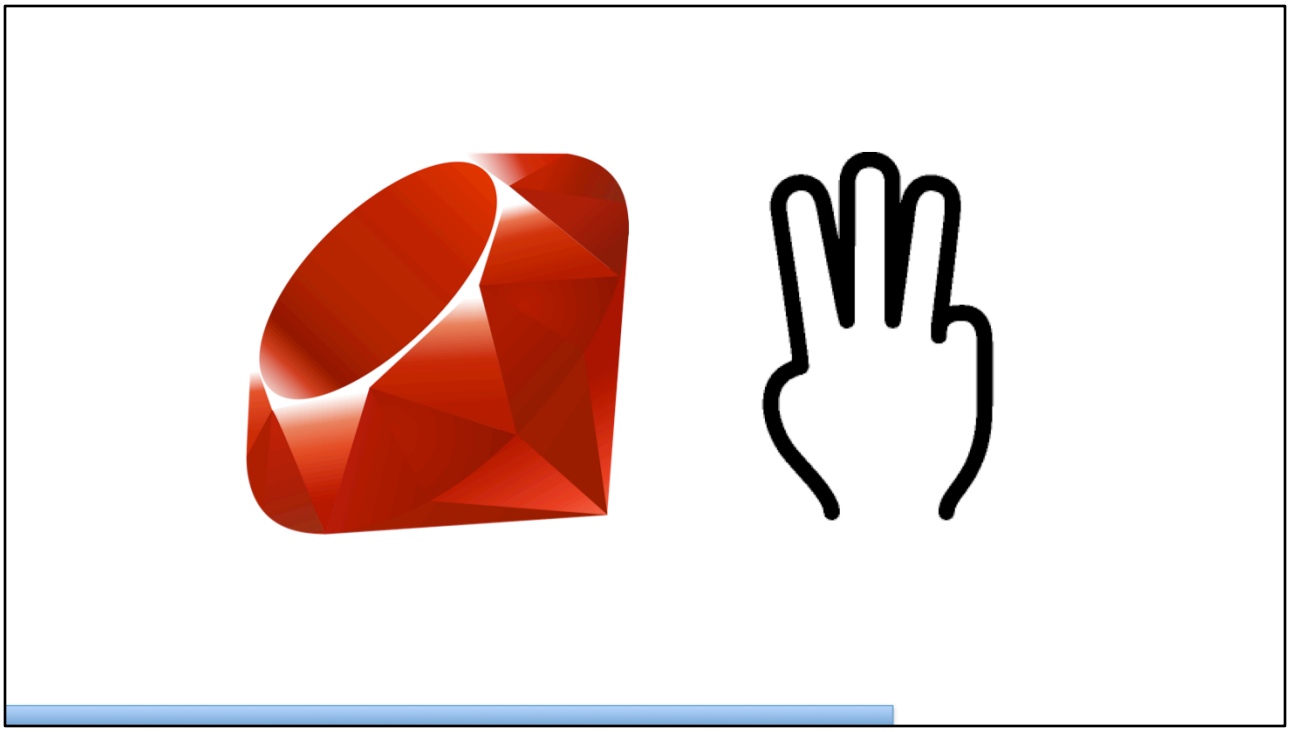


Ruby is optimized for programmer happiness.



So we try to optimize for Rubyist happiness.

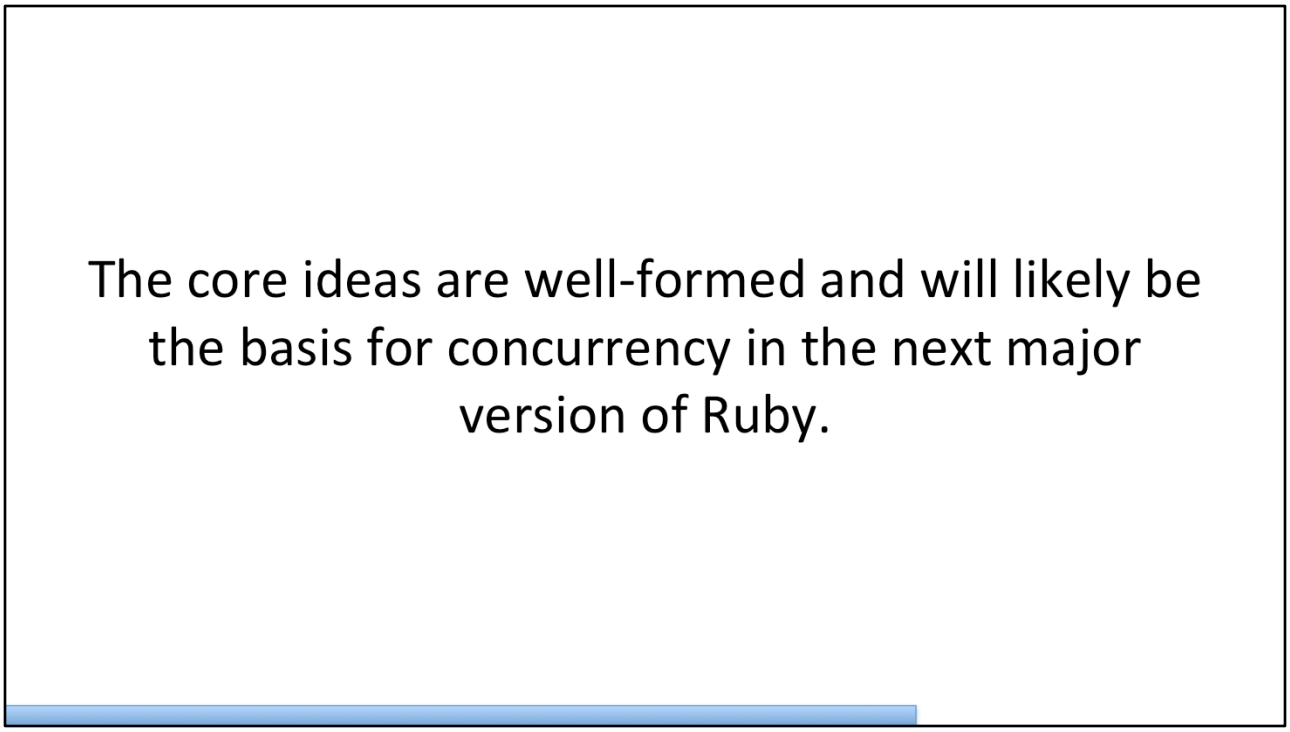
And I think we do a pretty good job.



But what about Ruby 3?

A concurrency model called “guilds” was recently proposed.

As I mentioned earlier.



The work has only been experimental. It is in the very earliest stages.
So many details still need worked out.

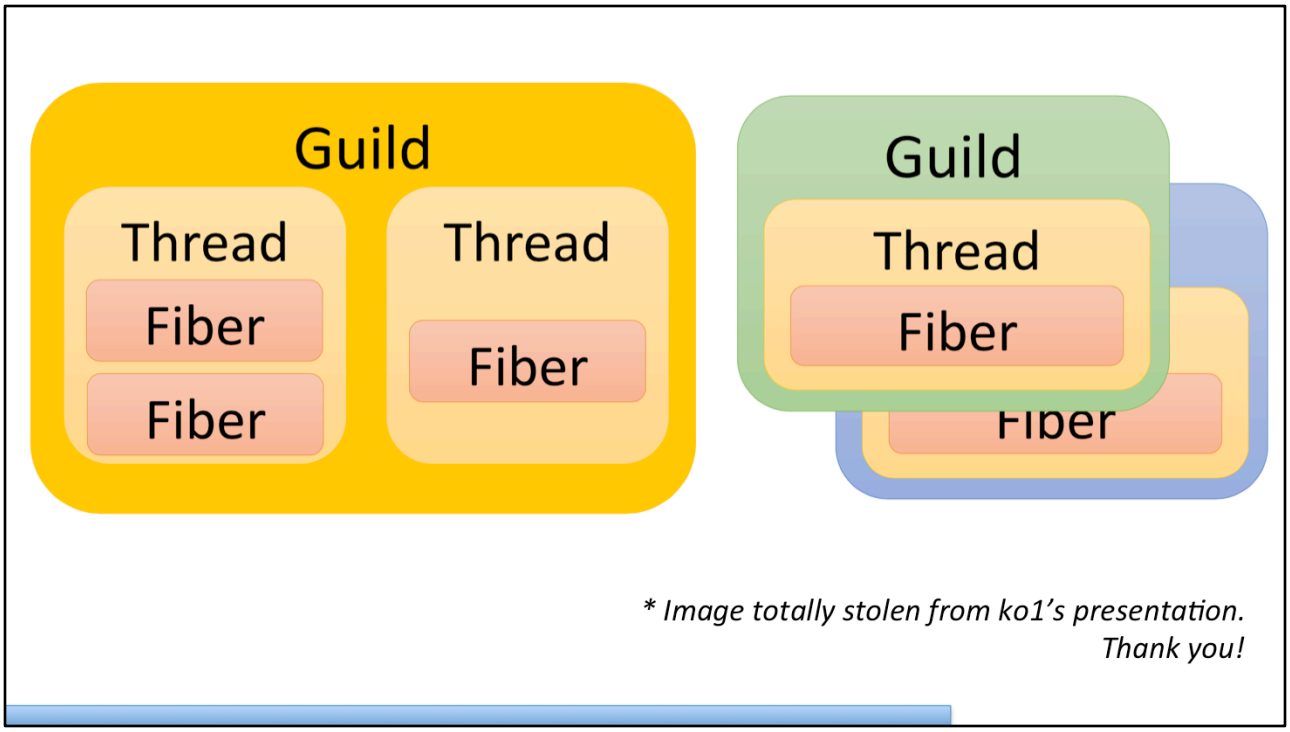
Not surprisingly, it does pretty much everything we've talked about so far.

But it does it in the language itself.

Quite simply, guilds...

Encourage proven designs.
Provide the strongest possible guarantees.
Embrace the Ruby language.

Sound familiar?



The core idea is highly influenced by Erlang, with a little inspiration from Go and Rust.

Every guild has at least one thread.
Every thread has at least one fiber.
And every program has at least one guild.

Like Erlang, guilds are isolated from one another.

Guilds cannot access references which are members of other guilds.

Guilds communicate exclusively through channels.

Channels are one-way, and each guild has one implicit channel.

References are passed between guilds through a channel.

The channel is pretty similar to an Erlang mailbox but without all the distributed systems overhead.
So it's also similar to a Go channel, except that it's implicit to the guild.
When we send a message what we really do is transfer a reference between guilds.

The semantics of channel operations are far closer to true message passing than anything Ruby has had previously.

References can be passed two ways: deep copy or membership transfer.

Deep copy is similar to what functional languages do.

Deep copy can be expensive, and with complex objects can lead to odd situations.

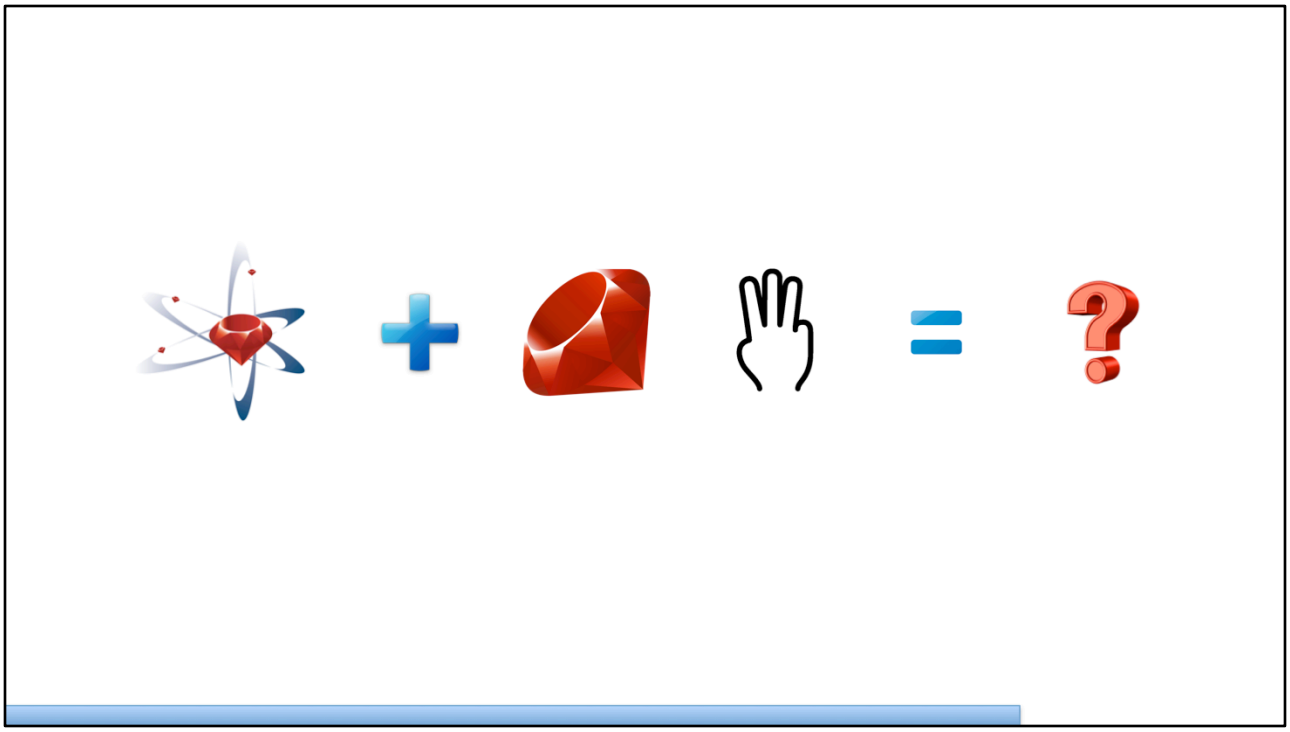
Remember, functional languages generally have value types, not object references.

The other things we can do is transfer guild membership.

When we transfer membership the object reference is invalidated in the sending guild.

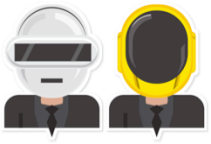
The object now belongs solely and fully to the receiving guild.

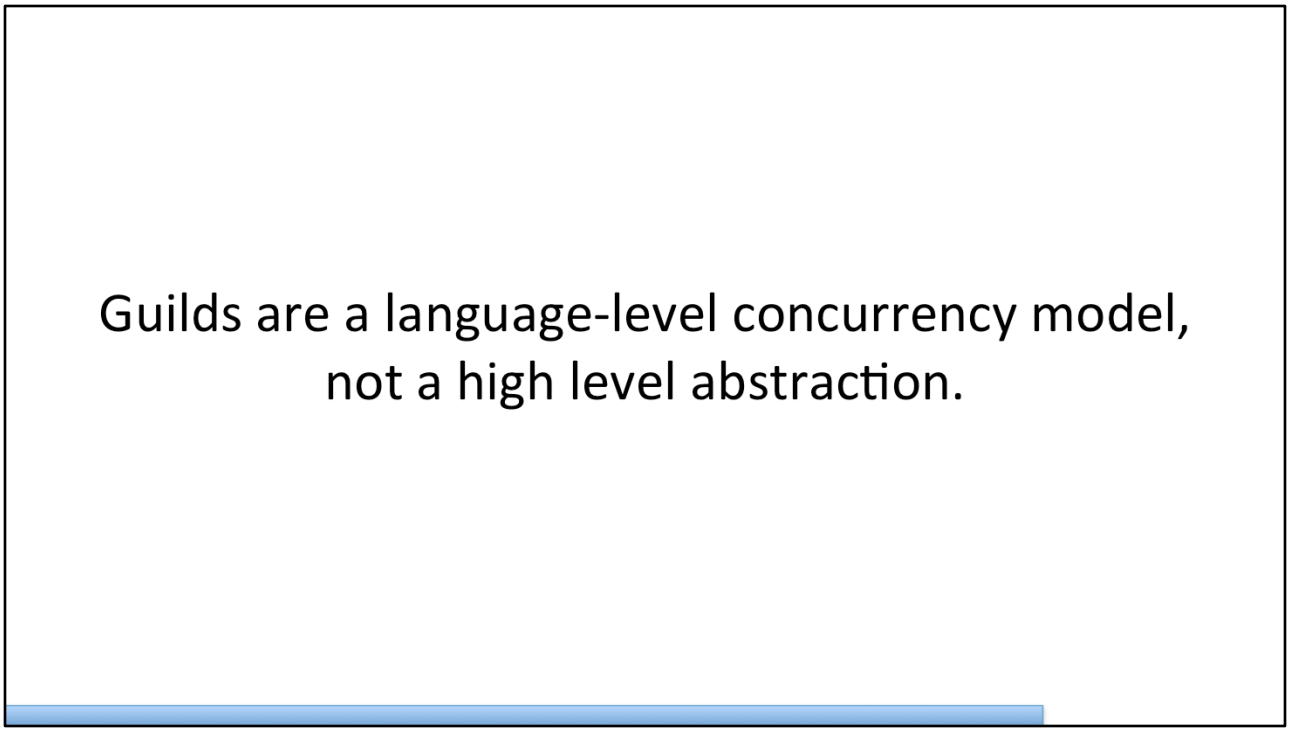
The other things we can do is transfer guild membership.
This is loosely inspired by Rust's ownership model (but only very loosely).
In this case we pass the reference, but we copy nothing.
This is true message passing. And it's entirely thread safe.



So what will happen to Concurrent Ruby when Ruby 3 arrives?

Nothing changes. We'll just get harder, better, faster, stronger.





Guilds are more like an Erlang process or a Golang goroutine than an Actor, Promise, Future, Task, etc.

Guilds provide a thread-safe, one-way communication mechanism between isolated threads of execution.

That's it.

This is very important. Critically important. But it's not enough for most programmers.

Guilds still require much work to be done by the application programmer.

Programmers must:

Design the interactions between guilds

Handle message receipt, probably in a loop.

Setup two-way communication for transferring results.

Handle exceptions

Communicate errors.

Programmers must do a number of things when writing concurrent applications.
…
Remember, Ruby is optimized for programmer happiness.
Doing all of these things every time I want to create a new guild doesn't feel like a happy path to me.

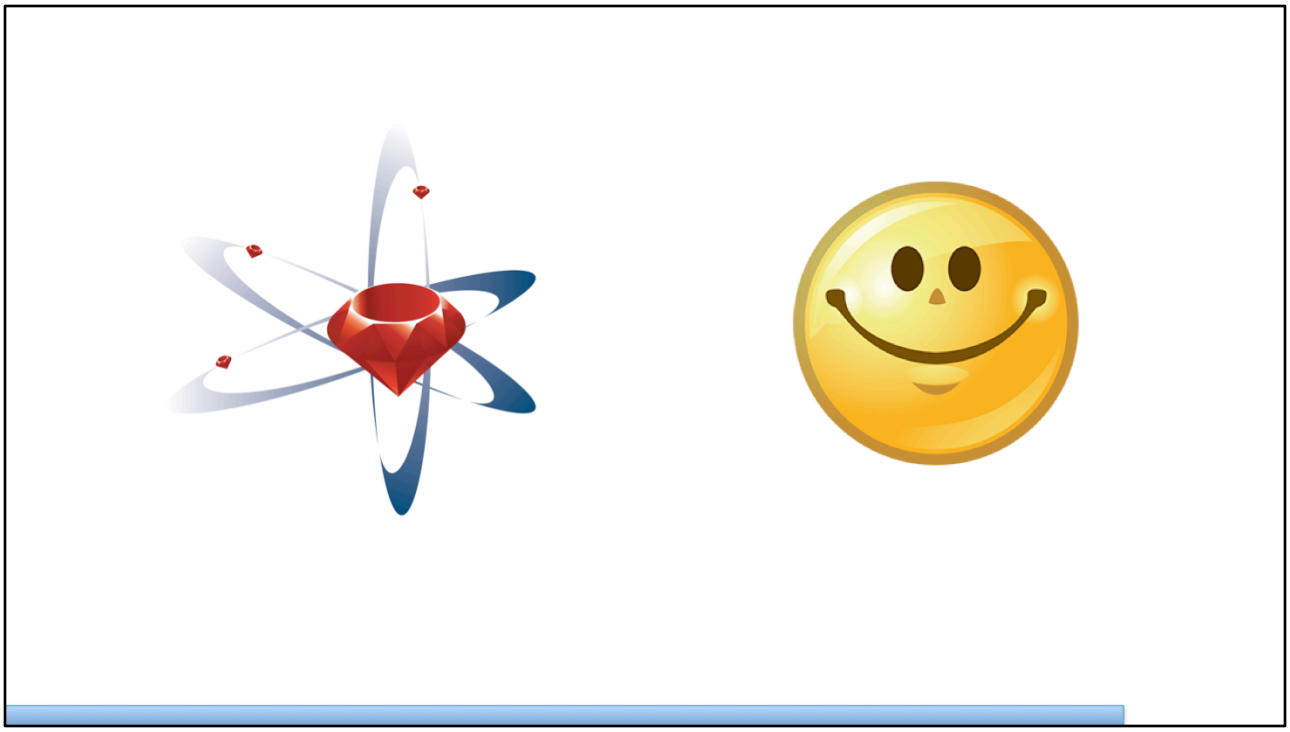
This is why most “concurrent” programming languages provide high-level abstractions in the standard library.

Erlang has gen_server, gen_event, gen_fsm, and supervisor.

Elixir has Agent and Task.

Scala has Akka.

And there are many other examples.



These things, and more, are already provided by Concurrent Ruby.

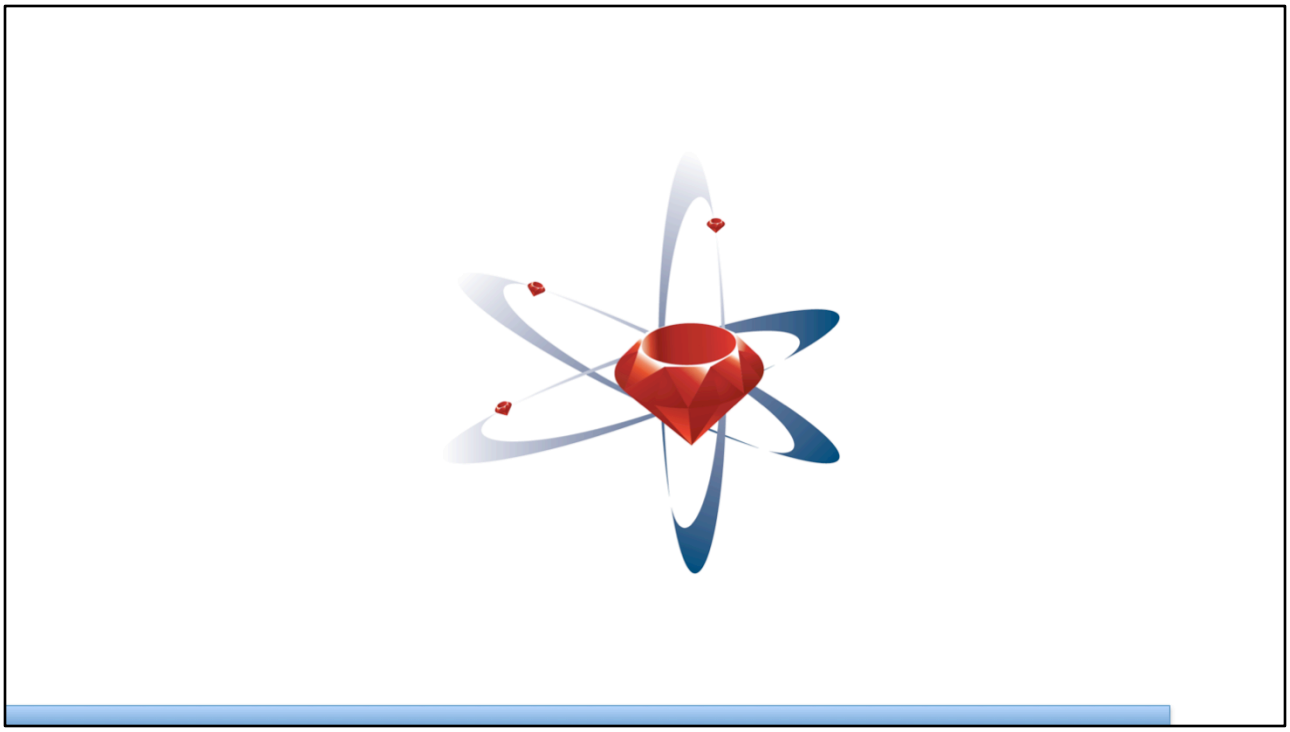
Guilds do not replace concurrency libraries like Concurrent Ruby.

Guilds give us better tools to build these libraries with.

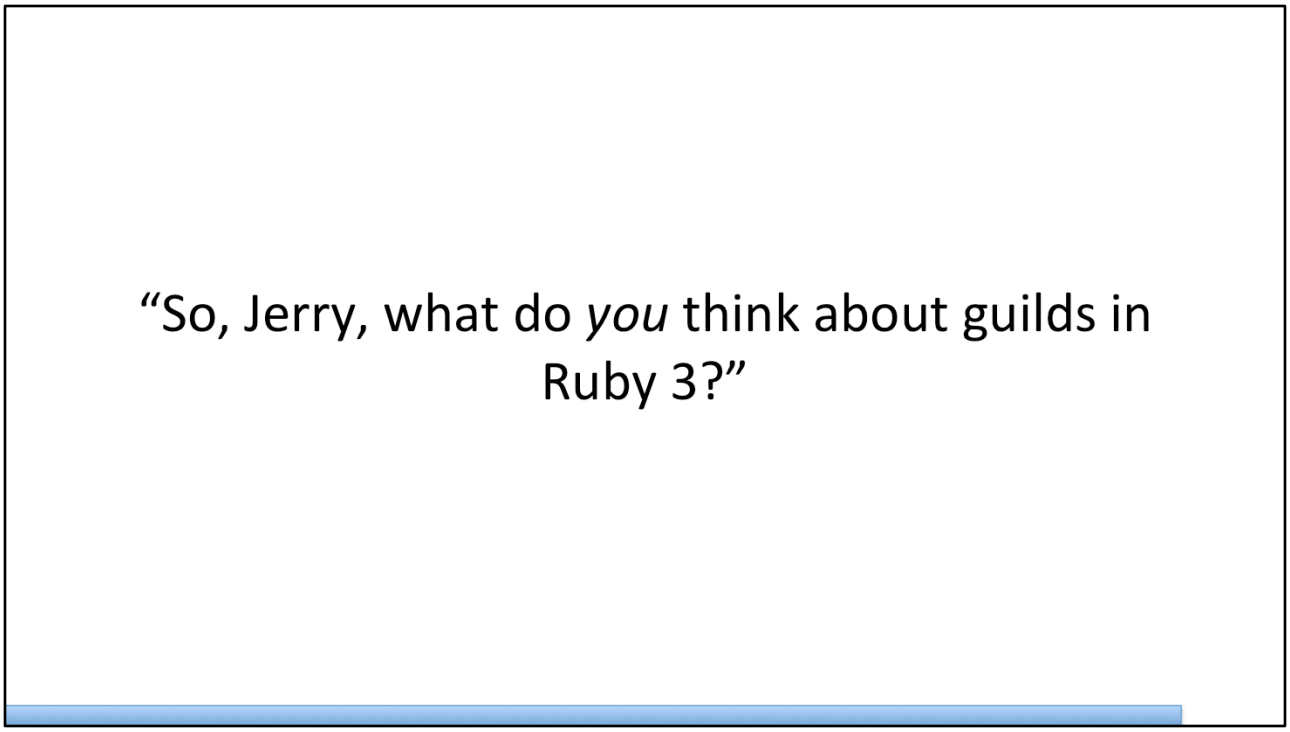
So what about you, the Ruby programmer? What should you do?

You want to build concurrent applications now.

But you also want to be prepared for Ruby 3.

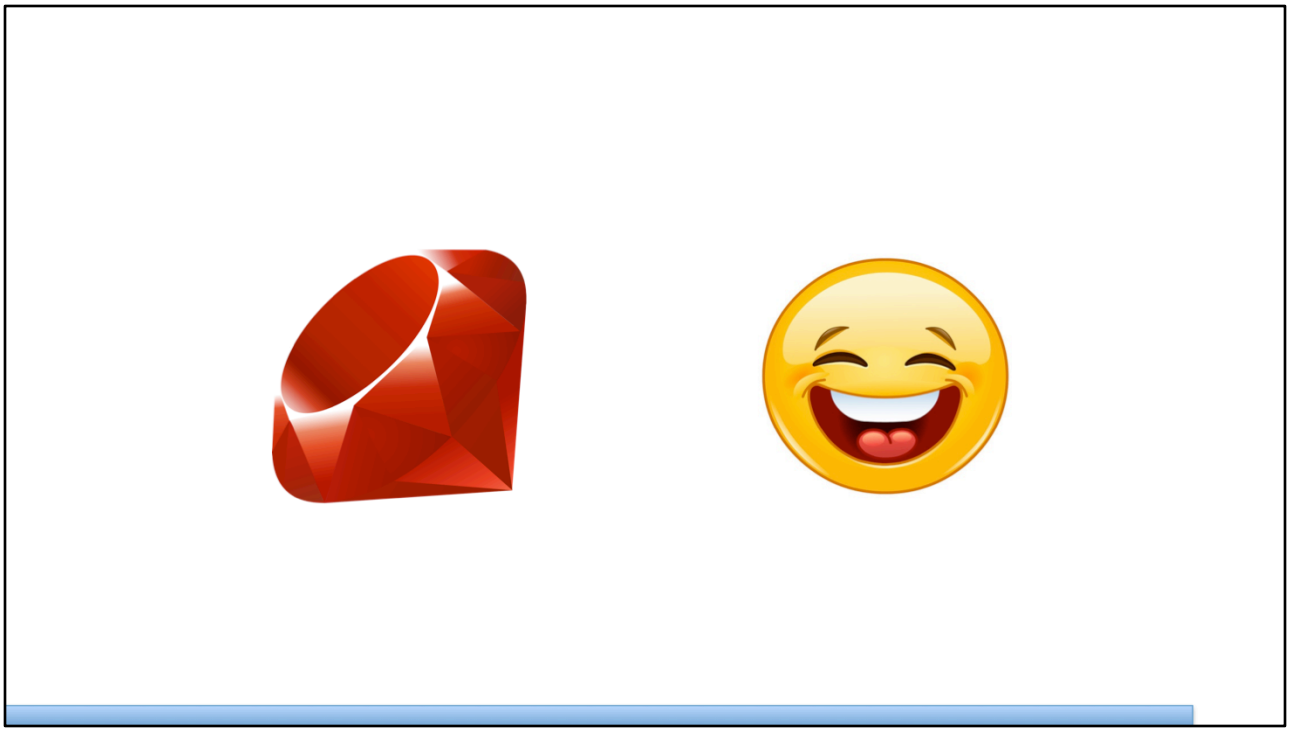


Simple. Use Concurrent Ruby. Or use any other high quality concurrency library.
Let us handle the Ruby 3 upgrade for you.
Once Ruby 3 becomes available we will update our internals to take advantage of the new concurrency features—guilds, channels, and message passing.
Our high-level abstractions will still work.
Your applications written today will still work.



Since the announcement of guilds, people have been asking me:

Well, thank you for asking. I’m happy to tell you what I think. :-)



Ruby makes me happy.

Guilds encourage proven concurrency practices.

Guilds provide strong guarantees and a formal memory model.

Guilds allow today's concurrency libraries, like Concurrent Ruby, to still exist.

But they give us better tools for simplifying our internals and provide stronger guarantees.

Guilds are heavily influenced by languages that are considered good at concurrency.

While embracing everything I love about Ruby.

Perhaps most importantly, do so…

This makes me very happy.

Again, my name is Jerry D'Antonio and I am from Akron, Ohio.

Obrigado!