Son of

Better Ruby Through Functional Programming

@deanwampler

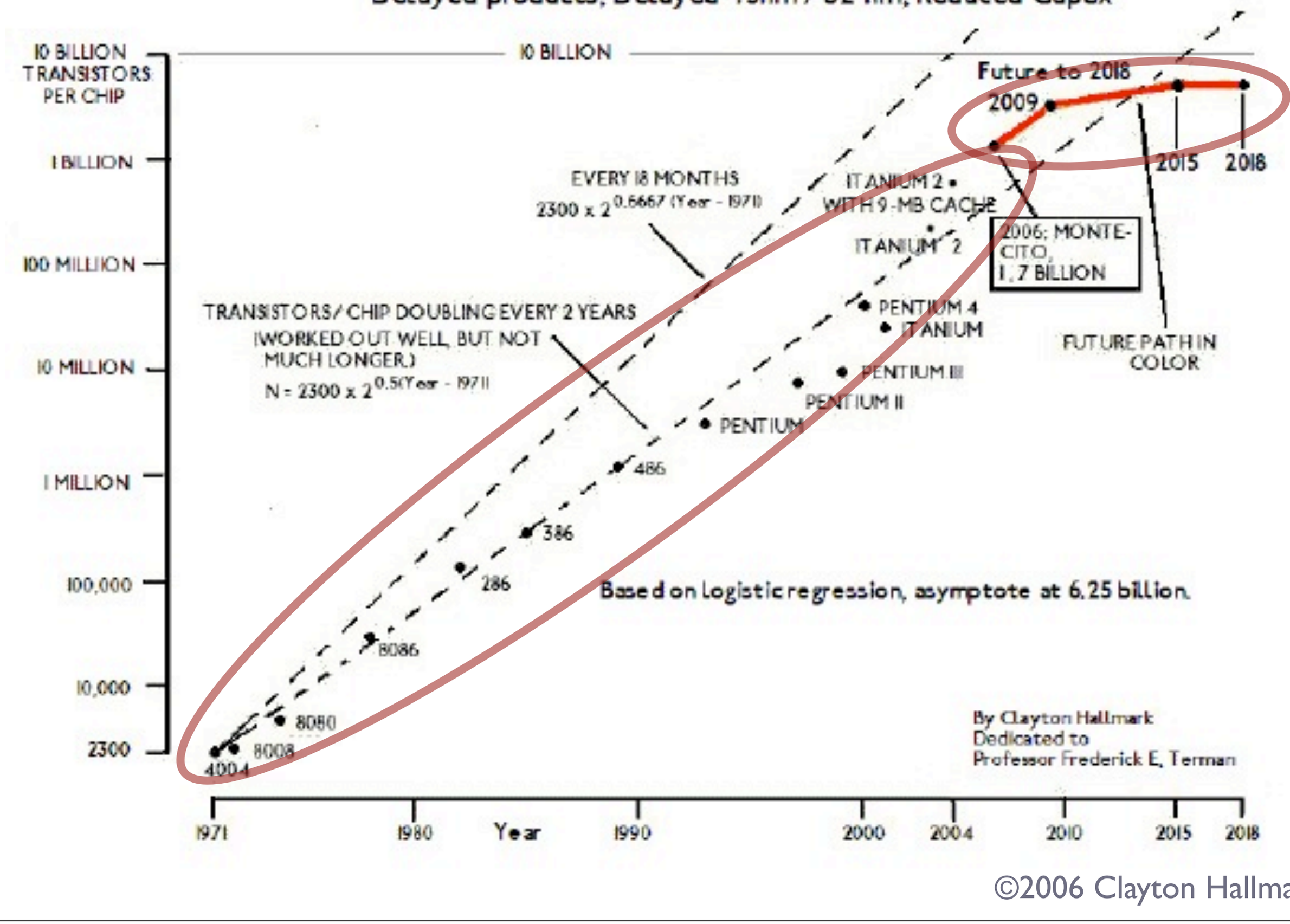
dean@objectmentor.com

polyglotprogramming.com/papers

This is the second version of the RubyConf2008 talk I did. Here, I'll summarize FP more quickly and dive into some specific application areas for FP.

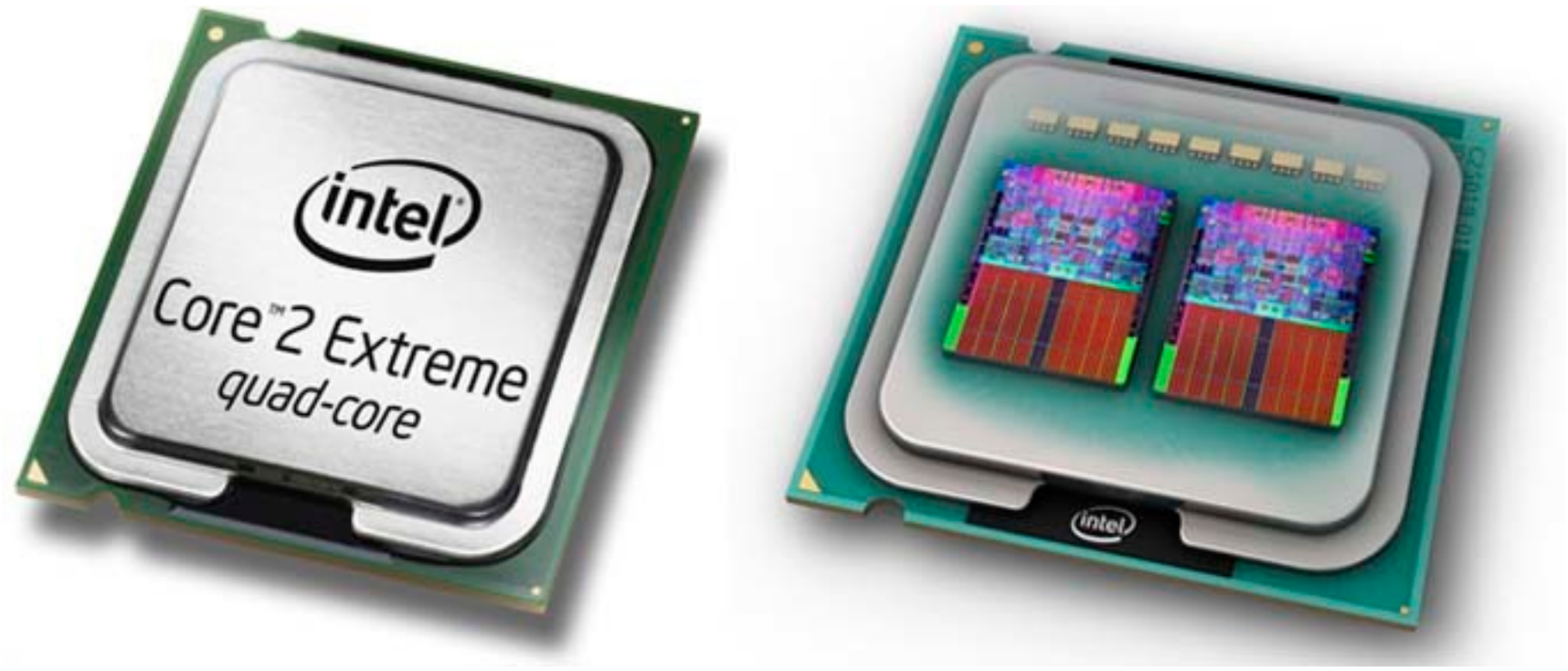
Why Functional Programming?

Why, after 50 years in academic and some industry labs is FP going mainstream??



Friday, September 11, 2009

We've hit the end of scaling through Moore's Law.



so, we’re scaling horizontally.

Single-Threaded Code Won't *Do*...

Friday, September 11, 2009

Single Threading can't scale anymore...

Friday, September 11, 2009

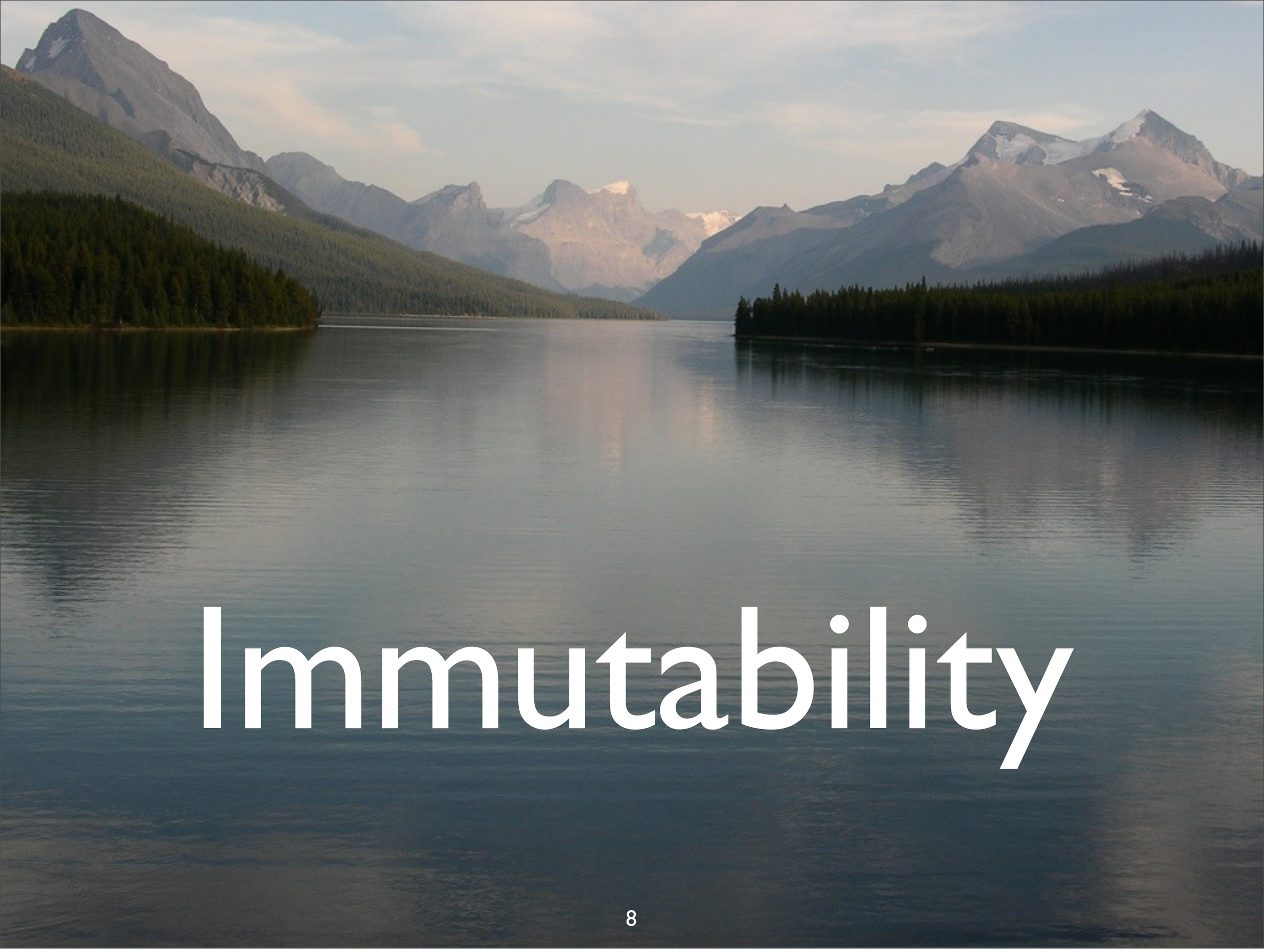
So we have to write concurrent code, but this is hard!!

"Functions", as in Mathematics

Functional Programming

Friday, September 11, 2009

"Functional Programming" is based on the behavior of mathematical functions and variables.



Friday, September 11, 2009

1st characteristic of FP: immutable values.

$$y = \sin(x)$$

Setting *x* fixes *y*

∴ *values* are *immutable*

Friday, September 11, 2009

"Functional Programming" is based on the behavior of mathematical functions.
"Variables" actually reference immutable values.

20 += x ??

We never *modify* the 20 "object"

"Functional Programming" is based on the behavior of mathematical functions.
"Variables" are actually immutable values.

Concurrency

No *mutable values*

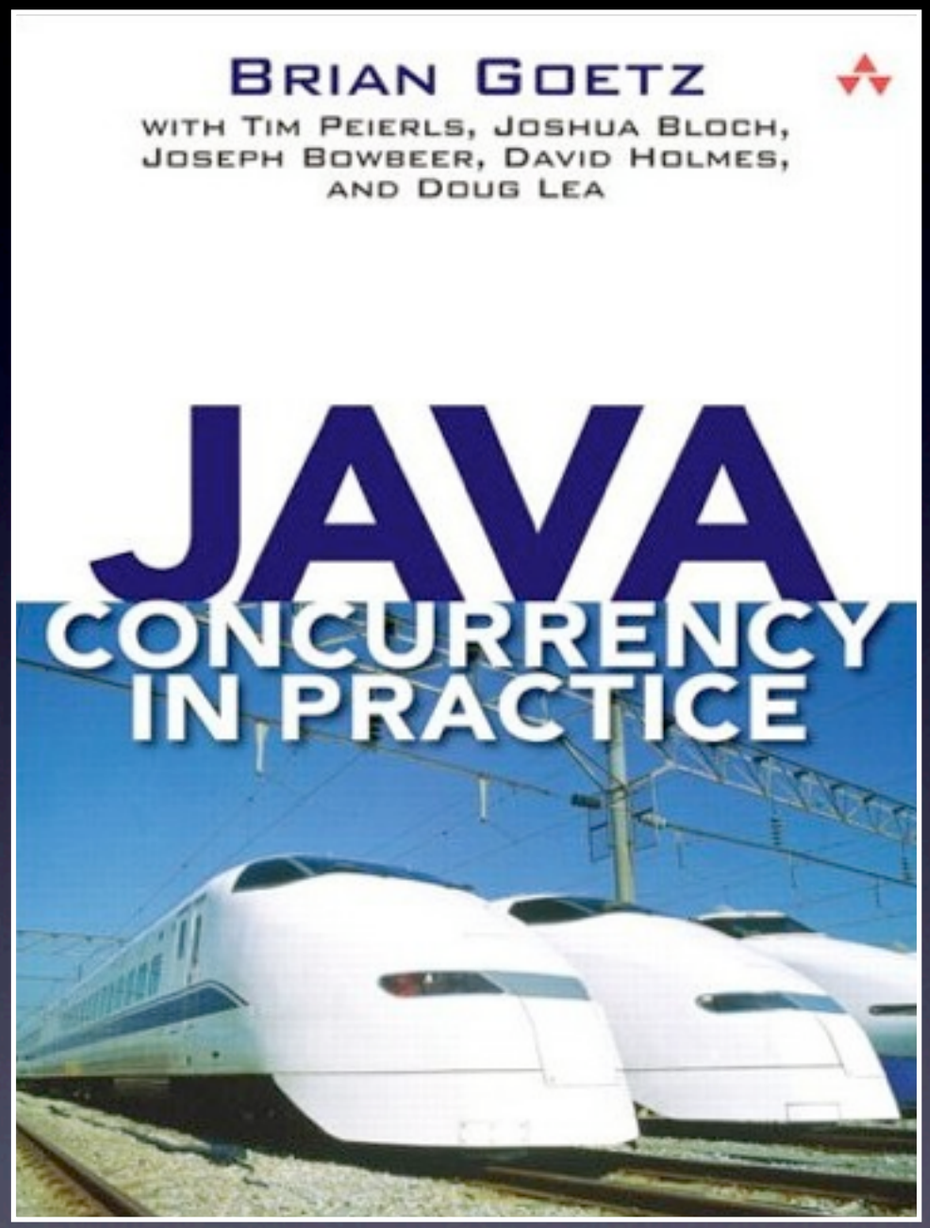
∴ *nothing* to *synchronize*

"No mutable values": There is changing state in a functional program; it's in the stack or new values returned by functions. Individual bits of memory are immutable.
FP is breaking out of the academic world, because it offers a better way to approach concurrency, which is becoming ubiquitous.

When you share mutable values...

Hic sunt dracones

(Here be dragons)



Friday, September 11, 2009

Your alternative, multi-threaded programming, which is very hard and few people can do it well.

Immutable *Types* in Ruby?

Friday, September 11, 2009

One way to be “pure” FP is to make your types immutable. (You can also “freeze” individual objects, but you must also freeze the attributes, recursively!)

```ruby
class Person
  attr_reader :first_name,
              :last_name,
              :addresses, …
  def initialize fn, ln, ads
    @first_name = fn.clone.freeze
    @last_name  = ln.clone.freeze
    @addresses  = ads.clone.freeze
  end
end
```

Read-only attributes

Array

Clone and freeze "subobjects"

First, we can support immutable types by using "attr_reader" only.
Don't forget to clone and freeze collections and other "subobjects". The string attributes have to be frozen, too. (We should actually clone and freeze each address itself, recursively.)

```
class Person
  …
  def update! other
    @first_name = other.first_name
    @last_name  = other.last_name
  …
end
end
```

Mutable!!

Use "!" methods with caution.

Second, don't allow self-modifying methods!

```
class Person
  …
  def update other
    Person.new other.first_name,
                other.last_name, …
  end
end
```

Better

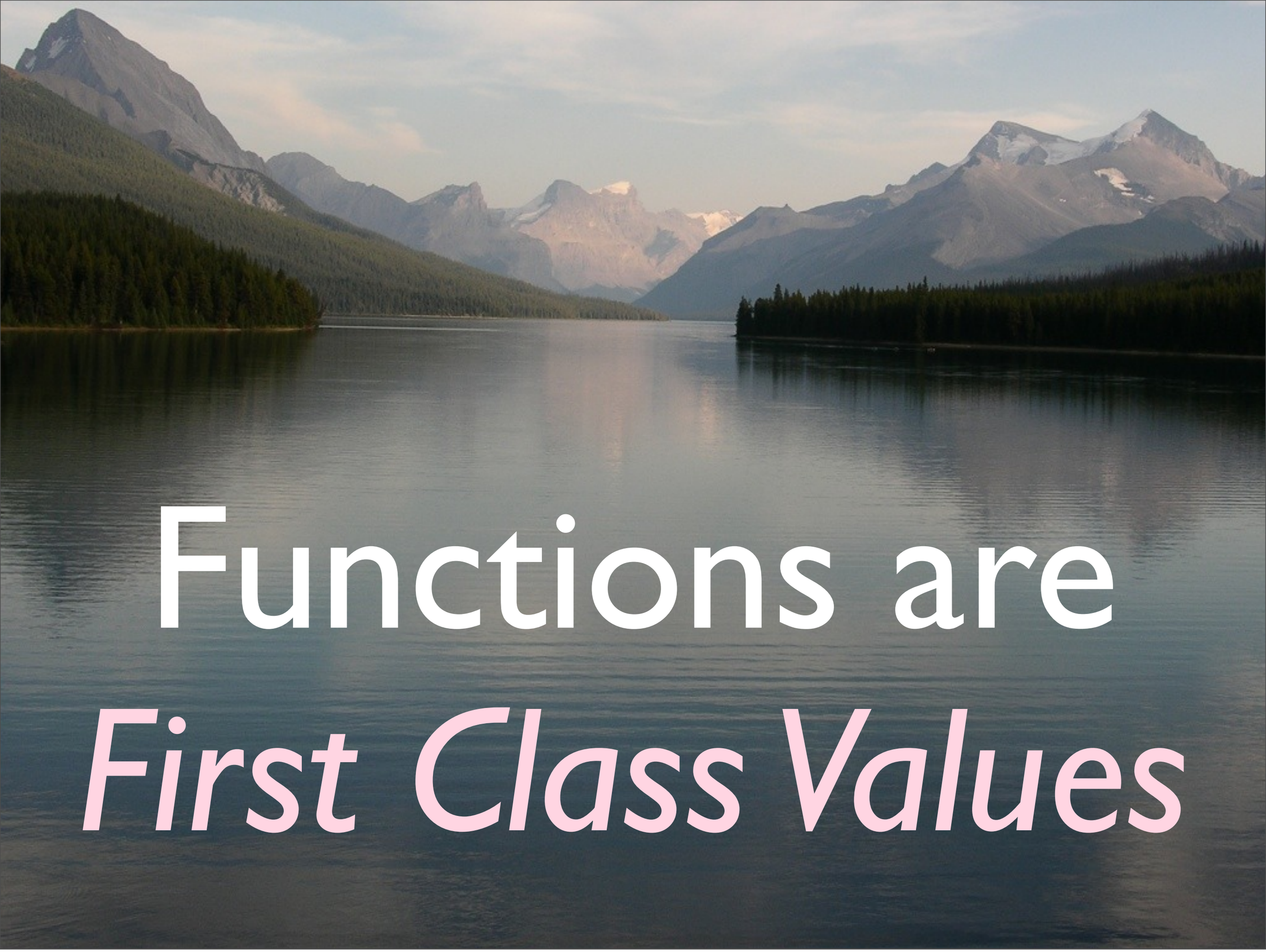
Returns a new Person

Instead, return a new instance.

Drawback: *Overhead* of Object *Creation*.

```
Hash#merge
vs.
Hash#merge!
```


You'll create lots of copies of objects if they are immutable. Big objects are expensive to copy.



2nd characteristic of FP.

tan(x) = sin(x)/cos(x)

Compose functions of other functions

∴ *first-class values*

Friday, September 11, 2009

Function are “first-class values”; you can assign them to variables and pass them to other (higher-order) functions, giving you composable behavior.


First-Class
Functions
in Ruby?

Is this a “function”?

```
def even?(n); n % 2 == 0; end

[1,2,3,4].find_all {|n| even? n}
# => [2, 4]
```

```
# The following don't work:
[1,2,3,4].find_all(even?)
[1,2,3,4].find_all(&even?)
```


Friday, September 11, 2009

You can’t just pass the method name…, even the “proc-ized” form, because of Ruby’s parse rules. Is it supposed to first invoke even? with no arguments and no parentheses and pass the result to find_all. Or, is it supposed to pass even? as a value? Ruby wants to call even? first.

Blocks, not *methods*, are *first-class* functions in Ruby.

```
[1,2,3,4].find_all {|n| even? n}
# => [2, 4]
```


So blocks/procs are the first-class “functions” of Ruby, but not methods.

3rd characteristic.

$$y = \sin(x)$$

Functions don't

change state

$\therefore$ *side-effect* free

Friday, September 11, 2009

A math function doesn't change any "object" or global state. All the work it does is returned by the function.

Side-effect free:

Easier to *understand* the *behavior*.

Friday, September 11, 2009

A side-effect free function is much easier to understand. You don't have to understand global or object state to understand its behavior. Functions that modify state are much "trickier".

Side-effect free:

Easy to *call* it *concurrently.*

Much less likely to have concurrency issues.

Side-Effect Free Functions in *Ruby*?

```ruby
class Person
  attr_reader   :first_name, …
  attr_accessor :addresses

  def initialize first_name, …
    …
  end
end

def filter array
  array.select {|s| yield s}
end
```

Which ones are side-effect free?

There's no mechanism to verify that a given method or block is side-effect free. You have to do it "manually", if you can. Note that construction of objects isn't side-effect free, but it's an exception we can live with, if we're careful! In a concurrent environment, "select" is side-effect free, but what if the "array" is modified by another thread? What if the implicit block has side effects?

But wait, there's more!

Recursion

Actor model of concurrency

Declarative vs. imperative

...

Pattern matching vs. polymorphism

Currying

There's 50 years of additional stuff we could talk about. We'll come back to a few bits later.

where the isospin operator I_T is

$$I_{T,M_T} = \begin{cases} 1 & T=0,\ M_T=0 \\ \tau_0=\tau_3 & T=1,\ M_T=0 \\ \tau_\pm=\frac{1}{2}(\tau_1 \pm i\tau_2) & T=1,\ M_T=\pm 1. \end{cases} \qquad (7.41)$$

Hence,

$$\langle \tfrac{1}{2}\|I_T\|\tfrac{1}{2}\rangle = \begin{cases} \sqrt{2} & T=0,\ M_T=0 \\ \sqrt{6} & T=1,\ M_T=0 \\ \mp\sqrt{3} & T=1,\ M_T=\pm 1. \end{cases}$$

Therefore, the matrix elements squared are of the form

$$\frac{1}{[J_I]^2}\sum_{M_I,M_F}\left|\sum_{J,M}\sum_{T=0}^{1}\sum_{(n)}\langle J_F M_F T_F M_{T_F}|O^{(n)}_{JM,TM_T}|J_I M_I T_I M_{T_I}\rangle\right|^2$$

$$= \sum_{J,M}\frac{1}{[J_I]^2[J]^2}\left|\sum_{T=0}^{1}\sum_{(n)}\langle \tfrac{1}{2}\|I_T\|\tfrac{1}{2}\rangle \begin{pmatrix} T_F & T & T_I \\ -M_{T_F} & M_T & M_{T_I}\end{pmatrix}\langle O^{(n)}_{JT}\rangle_M\right|^2, \qquad (7.42)$$

where the short-hand notation is used.

$$\langle O^{(n)}_{JT}\rangle_M \equiv \sum_{\alpha,\alpha'}\psi_{J,T}(\alpha,\alpha')\langle\alpha'|\hat{O}^{(n)}_{JT}\|\alpha\rangle_M, \qquad (7.43)$$

where $\langle\alpha'\|\hat{O}^{(n)}_{JT}\|\alpha\rangle_M$ is the ...ement, reduced in angular mo-
...dependence removed. Actually, *all* the isospin dependence
has not been removed because the current form factors are isospin dependent (see
discussion in Section 7.5. Hence the isospin index T is ...

The index (n) is used to ...

Fear not the mathematics...

You don't have to be a math wiz to benefit from functional thinking...

where the isospin operator I_T is

$$I_{T,M_T} = \begin{cases} 1 & T=0,\ M_T=0 \\ \tau_0=\tau_3 & T=1,\ M_T=0 \\ \tau_\pm=\frac{1}{2}(\tau_1 \pm i\tau_2) & T=1,\ M_T=\pm 1. \end{cases} \qquad (7.41)$$

Hence,

$$\langle \tfrac{1}{2}\|I_T\|\tfrac{1}{2}\rangle = \begin{cases} \sqrt{2} & T=0,\ M_T=0 \\ \sqrt{6} & T=1,\ M_T=0 \\ \mp\sqrt{3} & T=1,\ M_T=\pm 1. \end{cases}$$

Therefore, the matrix elements squared are of the form

$$\frac{1}{[J_I]^2}\sum_{M_I,M_F}\left|\sum_{J,M}\sum_{T=0}^{1}\sum_{(n)}\langle J_F M_F T_F M_{T_F}|O^{(n)}_{JM,TM_T}|J_I M_I T_I M_{T_I}\rangle\right|^2 \qquad (7.42)$$

$$= \sum_{J,M}\frac{1}{[J_I]^2[J]^2}\left|\sum_{T=0}^{1}\sum_{(n)}\langle \tfrac{1}{2}\|I_T\|\tfrac{1}{2}\rangle \begin{pmatrix} T_F & T & T_I \\ -M_{T_F} & M_T & M_{T_I}\end{pmatrix}\langle O^{(n)}_{JT}\rangle_M\right|^2,$$

where the short-hand notation is used.

$$\langle O^{(n)}_{JT}\rangle_M \equiv \sum_{\alpha,\alpha'}\phi_{J,T}(\alpha,\alpha')\langle\alpha'|\hat{O}^{(n)}_{JT}\|\alpha\rangle_M, \qquad (7.43)$$

where $\langle\alpha'\|\hat{O}^{(n)}_{JT}\|\alpha\rangle_M$ is the [illegible] matrix element, reduced in angular momentum, with the isospin dependence removed. Actually, *all* the isospin dependence has not been removed because the current form factors are isospin dependent (see discussion in Section 7.5. Hence the isospin index T is

The index (n) is used to

Apply its useful lessons.

Let's deep dive into some general lessons from FP.

More *Precise* Objects

Friday, September 11, 2009

In many mature code bases, I see ill-defined, hard-to-maintain object models. Can functional thinking help?

Consider an *algebraic* type:

Point

For 2-dimensional points

Friday, September 11, 2009

Let's look at 2-dimensional points for a minute.

```ruby
class Point
 attr_reader :x, :y
 def initialize x, y
  @x = x, @y = y
 end

 def + p2
  Point.new(@x + p2.x, @y + p2.y)
 end
 def - p2
  Point.new(@x - p2.x, @y - p2.y)
 end
end
```


A point class with immutable instances (assuming floats for x & y). To add them, you add the x values and the y values (similarly for subtraction). Also, note that they are immutable. Math operations create new instances.

```
describe "Point" do
 before :all do
  @samples =
   (-3..3).inject([]) do |l1,i|
    l1 +=
     (-3..3).inject([]) do |l2,i|
      l2 << Point.new(1.1*i,1.1*j)
    end
  end
end
...
```


Before all, create a list of samples, combinations of x and y values between (1.1 * -3) and (1.1 * +3).

```
describe "Point" do
  …
  it "should support + ..." do
    @samples.each do |p1|
      @samples.each do |p2|
        p12 = p1 + p2
        p12.x.should == p1.x + p2.x
        p12.y.should == p1.y + p2.y
      end
    end
  end  // “-” is similar
```


Then we test all the permutations to “ensure” that the algebraic properties hold for all instances (at least all that we try…).

For *all* instances,
certain *properties*
are *true*.

Data types in functional programs are typically like this.

Objects should have well-defined *states* and *state transitions*.

Friday, September 11, 2009

FP makes us think about how well defined our objects are. Do we always know the states they can be in? Are the state transitions (behaviors) well defined?

Example #2

Money

is also

algebraic.

Friday, September 11, 2009

Closer to home for some of you, if you've worked on financial apps.

Example #2

- Represent Money as:
  - A String?
  - A Float?
  - A BigDecimal?
  - or ...

Friday, September 11, 2009

For “simple” things, it’s tempting to use primitive types like Strings. There’s only one attribute (the value), right? But, is that the best idea?

```ruby
class Money
 def initialize v
  @value = v
 end

 protected
 attr_reader :value
```

Don't "leak" the implementation

```ruby
def to_s  # assume Dollars...
 format("$%.2f", value)
end

...
```


Ignores currency conversions, checking that the passed-in initial value is a float, accuracy - what should the type of @value be, etc.
Note that the value reader method is protected. We don't want the implementation to leak. (tests can use money.send(:value) ...)

```
...
# TODO: proper rounding...
def + other
 Money.new(value + other.value)
end

def - amount
 Money.new(value - other.value)
end

def * factor
 Money.new(value * factor)
end
...
```


Ignores financial standards for accurate rounding.

```
…
  include Comparable

  def <=> other
    # TODO: proper rounding...
    value <=> other.value
  end
end
```


Completes the minimum requirements for “algebraic” behavior. (We omitted this method from Point earlier, but it should be there.)

The *specs* are similar to those for Point.

Other Examples

- A Zip Code?
- A person's Age?
- An email or street Address?
- ...

Friday, September 11, 2009

For "simple" things, it's tempting to use primitive types like Strings. Is that the best idea?

Compose
domain models
from *precise*
building blocks.

Start with the building blocks, get them precise, then build up.

Bloated, Rigid Object Models

In many mature code bases, I see bloated, hard-to-maintain object models? Why and can functional thinking help?

Your Person

≠

My Person

How many apps/components in your systems have a different notion of “Person” (or any other common type)?

Use *one*, *big* Person type with all possible *fields*?

Friday, September 11, 2009

One implementation option ==> Bloat!!

Use *many, small* Person types?

Not bloated, but too ad hoc, gratuitously inconsistent?

Do we *need*
a Person type,
at all?

Does defining a type (or more than one) for Person deliver enough value to justify its existence?

But, first...

To consider answers to these questions, let's first look at one more aspect of functional programming...

Functional

Data Types

Friday, September 11, 2009

…, the classic functional data types and algorithms.

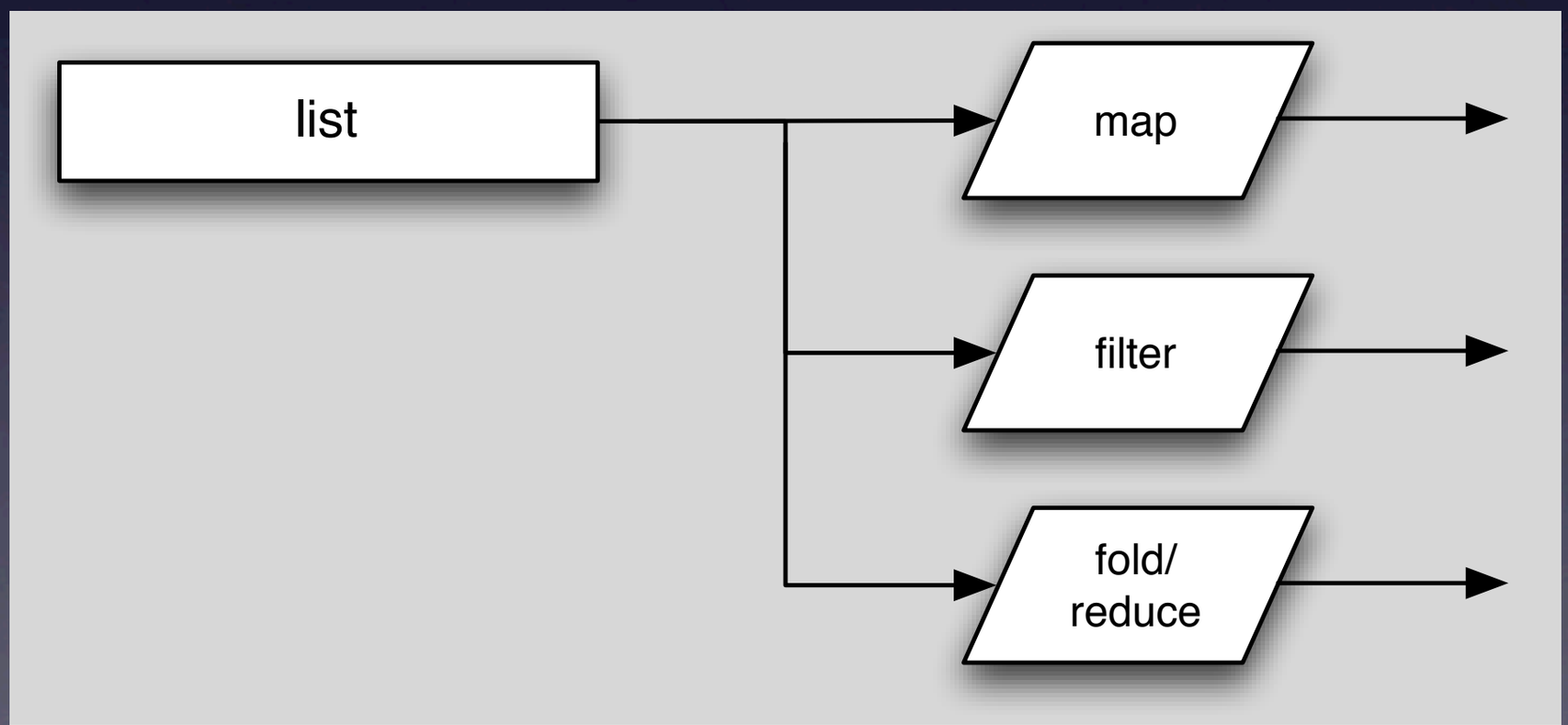
Lists and Maps (Hashes)

```
my_list = [
    "1 Memory Lane",
    "1 Hope Drive",
    "1 Infinite Loop"]
```

```
my_hash = {
    :addr1 => "1 Memory Lane",
    :addr2 => "1 Hope Drive",
    :addr3 => "1 Infinite Loop"}
```


Lists and (optionally) Maps are the most common functional data types. Some FP languages have others, like Sets.

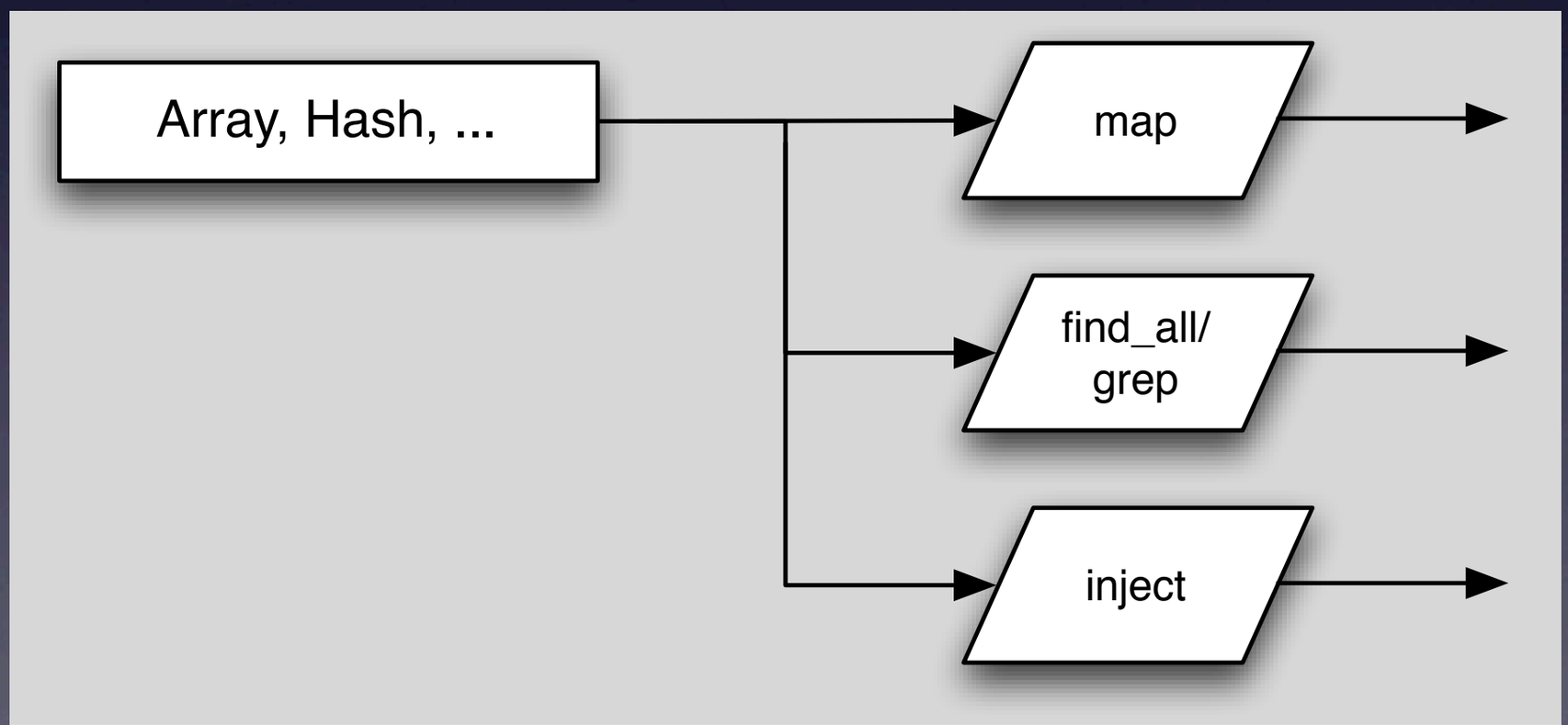
Classic Operations on *Functional Data Types*



Friday, September 11, 2009

List is the most commonly-used data type. Note that maps and sets can be represented as lists, too (ignoring performance issues from this implementation choice...).

In Ruby?



Ruby equivalents.

Back to:

Do we *need* a Person type, at all?

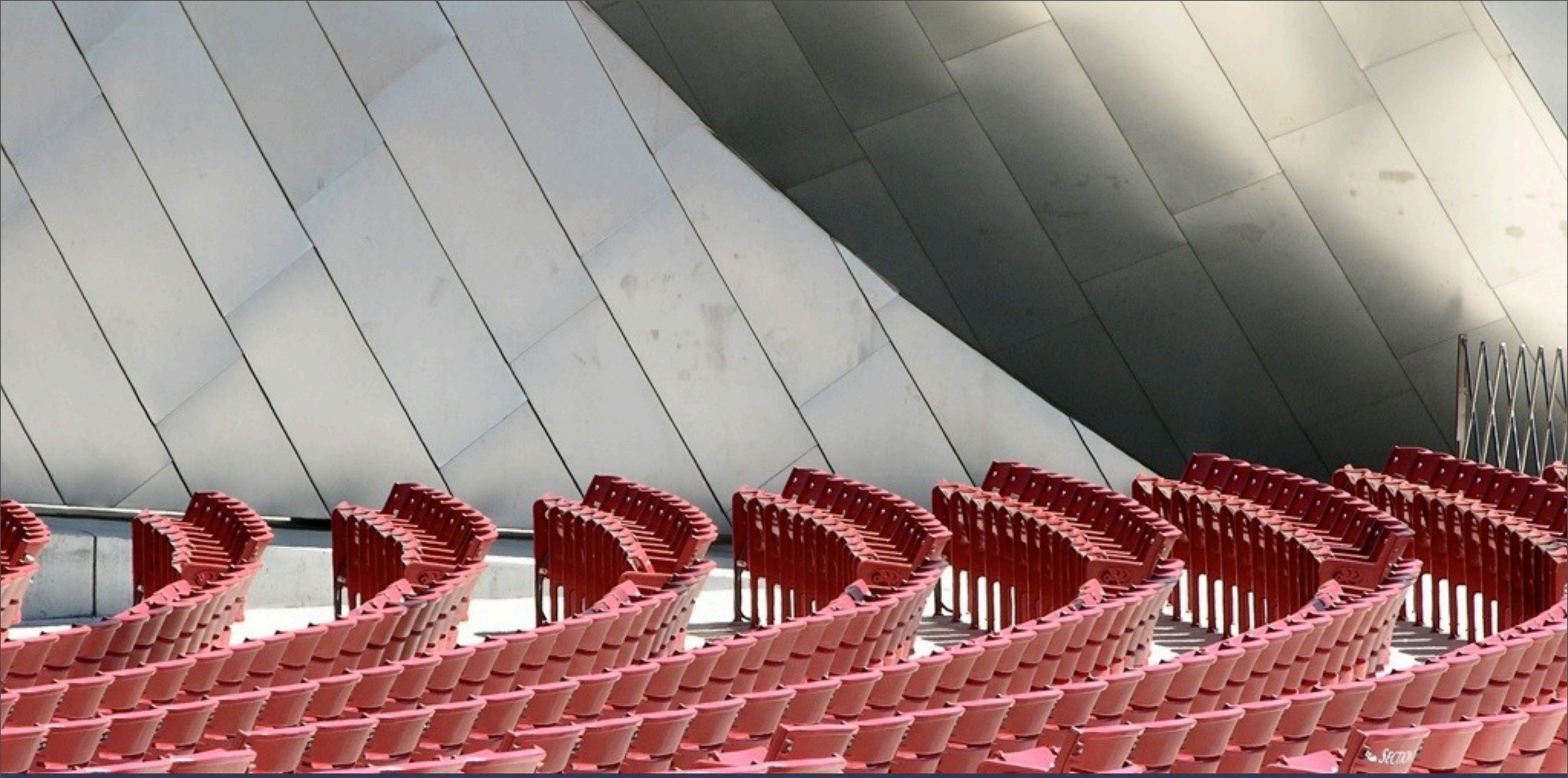
Maybe we're better off *without a type!*

If we have a lot of volatility in the definition of "Person", maybe trying to define a single or multiple Person classes is not really that beneficial. This is especially true if we're exchanging person data between components (as we'll see…).

```ruby
dean = {
   :first_name => "Dean",
   :last_name  => "Wampler",
   :addresses  => [
     "1 Memory Lane",
     "1 Hope Drive",
     "1 Infinite Loop"]}
```

```ruby
dean[:addresses].each do |addr|
  puts "Address: #{addr}"
end
```


Use a hash instead. Note the synergy with JSON. Could also use a Struct in a similar way. Other components/systems might use a different hash of values.



Software Components?

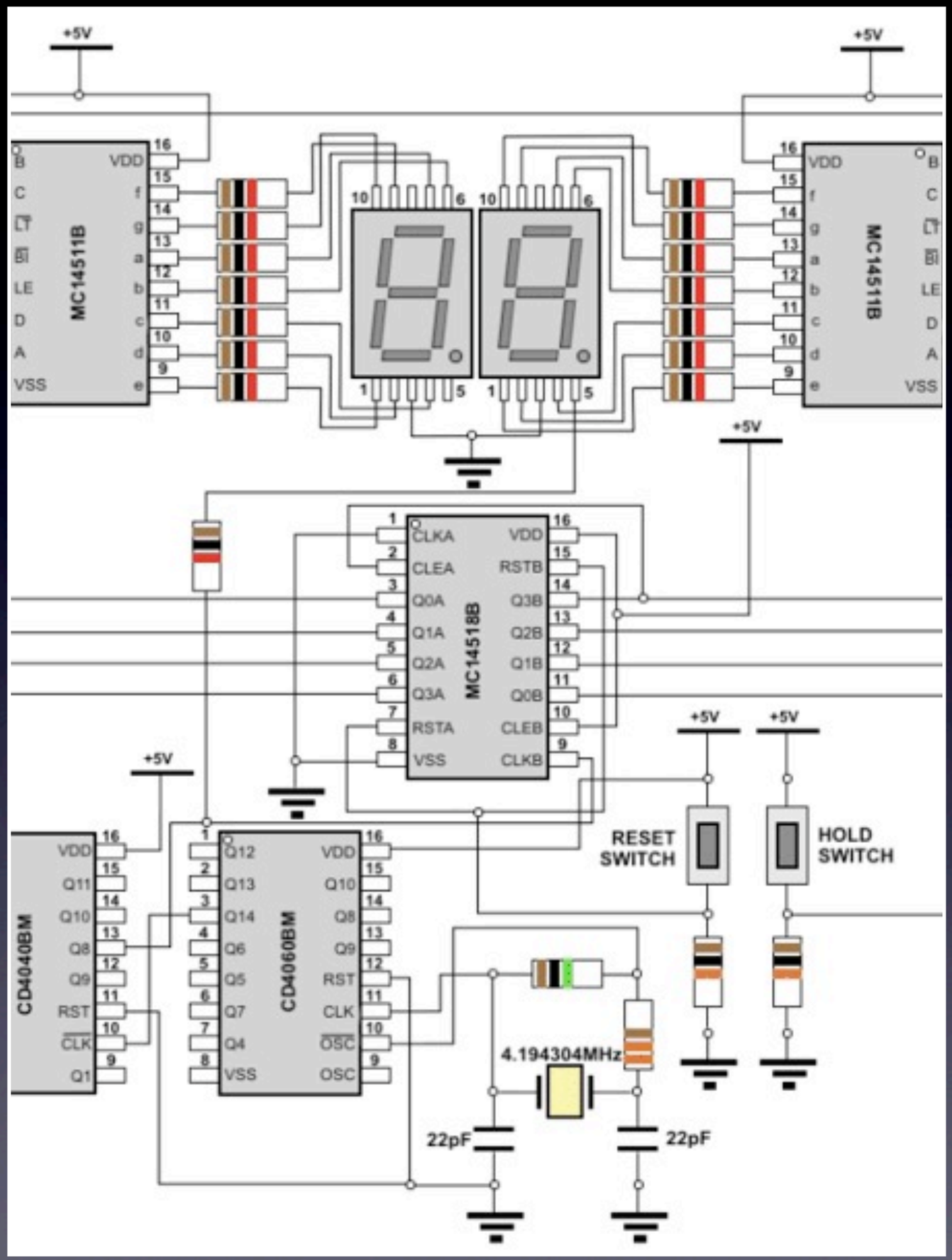
This leads us to the long-term goal of creating true SW components.

What makes a good *component model?*

. . . And why *didn't* objects *succeed* as *components*?

Examples of good *component models.*

Digital Electronics



Friday, September 11, 2009

With 2^N on/off signals, all the data and complex behavior of digital electronics has been possible.

HTTP

- Get
- Post
- Put
- Trace
- Connect
- Delete
- Head
- Options

WiFi/Ethernet, IP, TCP, HTTP, REST

HTTP – Very simple protocol. 8 messages, two of which are used for almost everything. Yet, that was enough to create the explosive growth of the Internet. (Of course, there are other protocols over Internet, like SMTP, IMAP, and POP, but HTTP drove adoption.)

Common Features

- Minimal, simple protocol.
- Straightforward encoding of higher-level information.

Features of successful component models.

Objects don't *meet* these *criteria!*

The richness and sophistication of objects has paradoxically undermined their utility as a component model!

A *Functional* Component Model

Friday, September 11, 2009

I suggest (and actually the FP community has said this for a long time…) that a functional-style model has more of the qualities I just described.

Data Is Represented by Lists and Maps

```
dean = {
   :first_name => "Dean",
   :last_name  => "Wampler",
   :addresses  => [
     "1 Memory Lane",
     "1 Hope Drive",
     "1 Infinite Loop"]}
```


We saw this already. Remember the JSON!!

Behavior is composed from the components' generic "primitives" and user-supplied blocks.

```
File.new("…").each_with_index
do |line, n|
 printf "%3d: %s", (n+1), line
end
```


A very simple example from the core library. The File "component" provides customization hooks through Enumerable and other methods. Note that it provides very generic, reusable behavior, but controlled ways to exploit those primitives.

Finally...

Finally...

Learn a *Functional* Language.

Thanks!

dean@objectmentor.com
@deanwampler

polyglotprogramming.com/papers
programmingscala.com