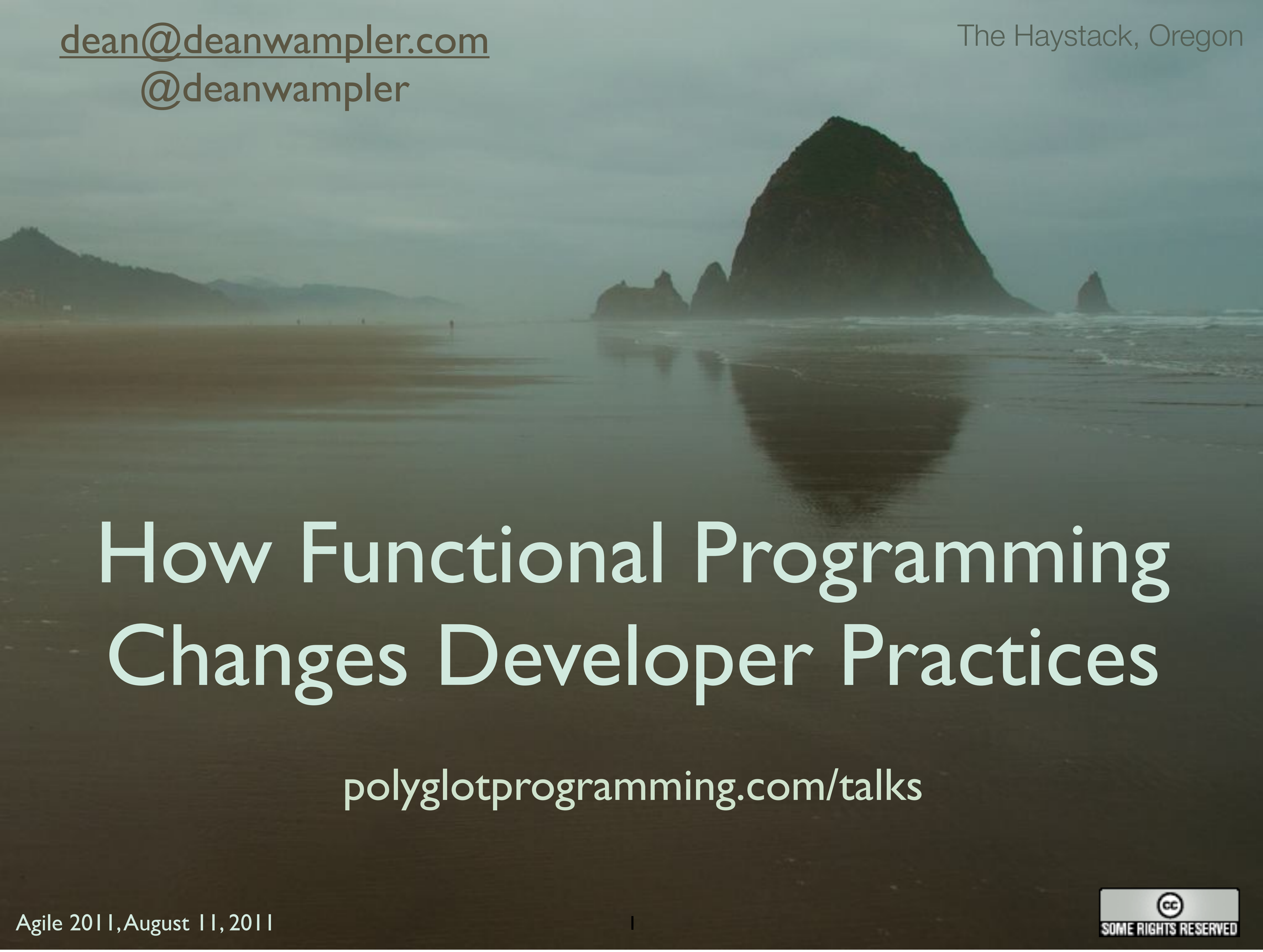
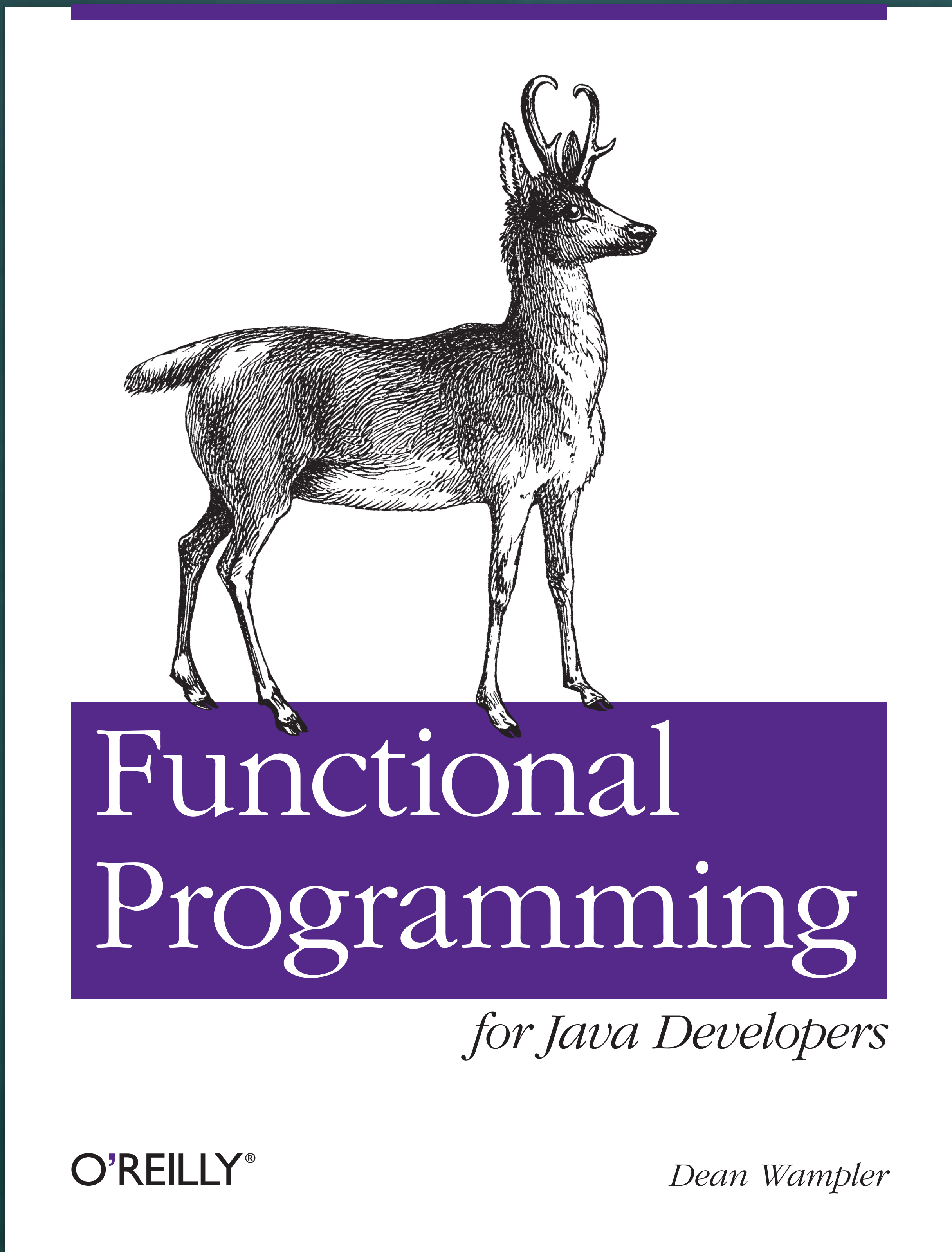
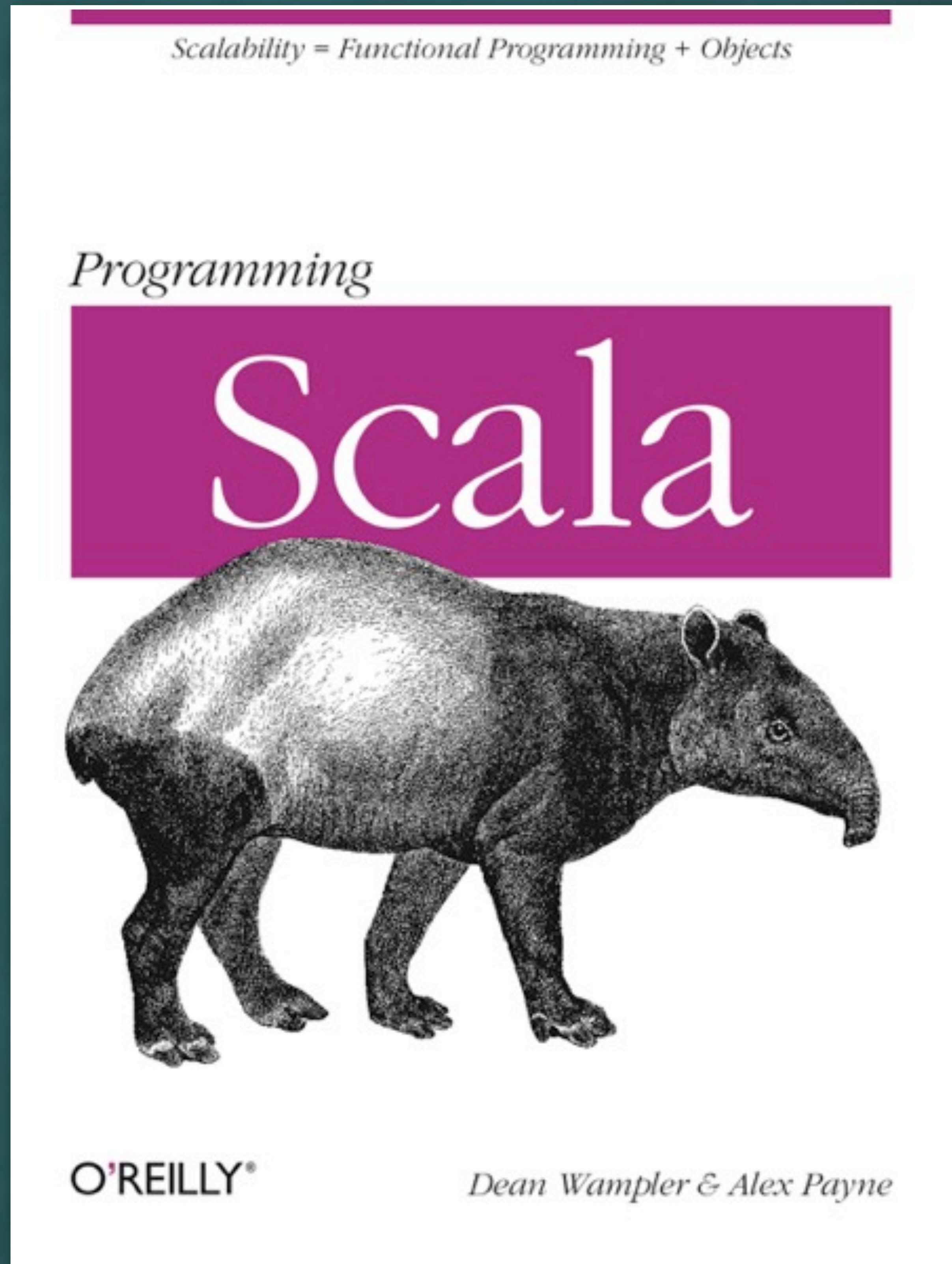


Friday, April 12, 13

Adapted from my longer tutorial at github.com/deanwampler/Presentations/BetterProgrammingThroughFP.
All photos © 2010 Dean Wampler, unless other noted. Most of my photos are here: http://www.flickr.com/photos/deanwampler/. Most are from the Oregon coast. Some are from the San Francisco area. A few are from other places I've visited over the years.

(The Haystack, Cannon Beach, Oregon)





polyglotprogramming.com/fpjava

programmingscala.com

Friday, April 12, 13

I got interested in FP about 5 years ago when everyone was talking about it. I decided it was time to learn myself and I expected to pick up some good ideas, but otherwise remain primarily an "object-oriented developer". Actually, it caused me to rethink my views and now I tend to use FP more than OOP. This tutorial explains why.

Friday, April 12, 13

Outline. I won't have time to talk about concurrency. See my github.com/deanwampler/Presentations/ BetterProgrammingThroughFP tutorial for more details.

(Nehalem State Park, Oregon)

What problems motivate the need for change, for which Functional Programming is well suited?

(Nehalem State Park, Oregon)

Concurrency is the reason people started discussing FP, which had been primarily an academic area of interest. FP has useful principles that make concurrency more robust and easier to write.

(San Francisco Bay)

We're Drowning in Data.

Google

twitter

facebook

YouTube

...

Not just these big companies, but many organizations have lots of data they want to analyze and exploit.

(San Francisco)

Friday, April 12, 13

I will argue that objects haven't been the modularity success story we expected 20 years ago, especially in terms of reuse. I'm referring to having standards that actually enable widespread interoperability, like electronics, for example. I'll argue that object abstractions are too high-level and too open-ended to work well.

(Mud near Death Hollow in Utah.)

We need better agility.

Schedules keep getting shorter. The Internet weeded out a lot of process waste, like Big Documents Up Front, UML design, etc. From that emerged XP and other forms of Agile. But schedules and turnaround times continue to get shorter.

(Ascending the steel cable ladder up the back side of Half Dome, Yosemite National Park)

Friday, April 12, 13

Every now and then, we need to stop, look at what we’re doing, and remove the cruft we’ve accumulated. If you’re a Java programmer, recall how efforts like the Spring Framework forced a rethinking of J2EE. I claim that a lot of the code we write, specifically lots of object middleware, is cruft. Functional programming isn’t *simple*, but in my view it reflects a refocusing on core principles and minimally-sufficient design.

(Maligne Lake, Near Jasper National Park, Jasper, Alberta)

What is Functional Programming?

Nehalem State Park, Oregon

Friday, April 12, 13

This is rich field, so I can't cover everything. I'll mention the things that I believe are most useful to know for beginners and those curious about FP.

(Nehalem State Park, Oregon)

Functional Programming

is inspired by

Mathematics.

FP follows the “rules” for the behavior of functions, variables, and values in mathematics. Everything else falls out from there...

Friday, April 12, 13

First, values in FP are immutable, but variables that point to different values, aren’t.

Immutable Values

```
y = sin(x)
1 = sin(π/2)
```

x and y are *variables*.

Once you assign a *value* to x,
you fix the *value assigned to* y.

First, values in FP are immutable, but variables that point to different values, aren't.

(At dusk flying over the Midwest – lightened)

Immutable Values

```
y = sin(x)
```

You can start over with new *values* assigned to the same *variables*.

But you never modify the *values*, themselves.

Immutable Values

π += 1

What would that mean?

Friday, April 12, 13

This would make no sense.

Immutable Values

If a value is *immutable*,
synchronizing access is no longer necessary!

Concurrency becomes far easier.

Of course, you don't need functional programming to make values immutable.

Friday, April 12, 13

Math functions don’t have side effects. They don’t change object or global state. All work is returned and assigned to y.

Functions

```
y = sin(x)
```

`sin(x)` does not *change state* anywhere!

Math functions don't have side effects. They don't change object or global state. All work is returned and assigned to y.

Referential Transparency

$$1 = \sin(\pi/2)$$

We can replace `sin(π/2)` with `1`.

We can replace `1` with `sin(π/2)`!

Functions and *values* are interchangeable

A crucial implication of functions without side effects us that functions and values are interchangeable. A mundane benefit is that it's easy to for an implementation to cache previous work for a given input value, for efficiency. But there are more profound benefits.

Functions

```
y = sin(x)
```

`sin(x)` can be used *anywhere*.
I don't have to worry about the
context where it's used

This makes testing, reuse, and concurrency much easier if I don't have to worry about external state modifications.

What is Functional Programming?

First-class functions

First Class Functions

```
i = 1
l = List.new(i, …)
f = lambda do |x|
  puts "Hello, #{x}!"
end
```

First Class: values that can be assigned to variables, pass to and from functions.

Lambda is a common name for *functions*.

A "thing" is first class in a language if you can use it as a value, which means you can assign it to variables, pass it as an argument to a function and return it from a function. In Ruby, objects, even classes are first class. Methods are not. Lambdas are ruby's way of defining anonymous functions (A second mechanism, Procs, is similar).
The term "lambda" comes from Lambda Calculus, a mathematical formalism developed in the '30s that explored how functions should work. The lambda symbol was used to represent anonymous functions.

We'll see the power of *First-class functions* in a moment...

We'll see how first-class functions let us build *modular*, *composable*, and *reusable* tools.

Friday, April 12, 13

(Nehalem State Park, Oregon)

I want to make the case that functional concepts lead to better modularity than objects.
Let’s look at one of the functional data structures, List, which we’ve already looked at a bit, but we need to explore further.

List

```
class List
  attr_reader :head, :tail
  def initialize(head, tail)
    @head = head
    @tail = tail
  end
  …
end
```

Head is the first *element*.

Tail is itself a List.

So, don't use attr_accessor or attr_writer in Ruby.
If you don't like dynamic typing, at least appreciate the compact, clean syntax.

Ruby

```
list = List.new(1,
         List.new(2,
          List.new(3, EMPTY)))
```

We need a special *tail* to terminate a List.

Creating a list (we'll see less verbose syntax later)
How should we terminate this list?? What should the special tail EMPTY be?? We'll come back to that.

List (cont.)

```
class List
  …
  def to_s
    "(#{head},#{tail})"
  end
  …
end
```


Note that to_s is recursive, because tail is a list and tail.to_s is called!

A separate *object* to represent *empty*.

```
class List
  …
  EMPTY = List.new(nil,nil)
  def EMPTY.head
    raise "EMPTY list has no head!!"
  end
  def EMPTY.tail
    EMPTY
  end
  def EMPTY.to_s
    "()"
  end
end
```


We declare a *constant* named EMPTY, of type List. We use nil for the head and tail, but they will never be referenced, because we redefine the head method for this “singleton” object to raise an exception, while tail simply returns EMPTY itself! We also define to_s to return “()”.
By overriding the methods on the instance, we’ve effectively given it a unique type.
(There’s a more short-hand syntax for redefining these methods, but for simplicity, I’ll just use the syntax shown.)
NOTE: It would be reasonable for EMPTY.tail to throw an exception like head throws.

```
class List
 …
 def to_s
  "(#{head},#{tail})"
 end
 …
 def EMPTY.to_s; "()"; end
 …
end
```

`List.to_s` is recursive, but `EMPTY.to_s` will terminate the recursion with *no conditional test!*

No conditional test is required in to_s to terminate the recursion. It is not an infinite recursion, though, because all lists end with EMPTY, which will terminate the recursion.
We've replaced a conditional test with structure, which is actually a classic OO refactoring.

List.to_s

```
puts List.new(1,
         List.new(2,
          List.new(3, EMPTY)

=> "(1,(2,(3,())))"
```

Lists are represented by *two* types: List and EMPTY.

For functional linked lists, only two types are used to represent all of them, List and EMPTY. That let us use the structural difference to manage recursion without conditional tests, among other benefits. We used nil to declare EMPTY, but never used those values.

Friday, April 12, 13

Let’s look at the 3 fundamental operations on data structures and understand their power.

Filter, map, fold

`filter`	Return a new collection with some elements removed.
`map`	Return a new collection with each element transformed.
`fold`	Compute a new result by accumulating each element.

All take a *function* argument.

The function argument tells each method what to do.

In Ruby...

filter	find_all
map	map
fold	inject

These names are not always used in different languages. Java doesn't even have these concepts in its collections! However, some 3rd-party libraries provide them.

Add map to List

f takes one arg, each item, and returns a new value for the new list.

```
def map(&f)
 t = tail.map(&f)
 List.new(f.call(head), t)
end
def EMPTY.map(&f); self; end
```

`f.call(head)` converts `head` into something new.

Add map first, because it’s the easiest. Note that we will show the implementations for both List and EMPTY together, to compare and contrast and to make the behavior of the recursion clear.

Example of map

```
list = … # 1,2,3,4
lm = list.map {|x| x*x}
puts "list: #{list}"
puts "lm:   #{lm}"
# => list: (1,(2,(3,(4,()))))
# => lm:   (1,(4,(9,(16,()))))
```


Demonstrate mapping a list of 4 integers to their squares. Note that we didn't modify the original list.

Add filter to List

f takes one arg, each item,
and returns true or false.

```
def filter(&f)
  t = tail.filter(&f)
  f.call(head) ?
   List.new(head, t) : t
end
def EMPTY.filter(&f); self; end
```

`f.call(head)` returns
true or false (keep or discard)

f.call(head) returns true if we keep the element or false if we discard it. If true, we return a new list with head and whatever t is. Otherwise, we just return t.

Example of filter

```
list = … # 1,2,3,4
lf = list.filter {|x| x%2==1}
puts "list: #{list}"
puts "lf:   #{lf}"
# => list: (1,(2,(3,(4,()))))
# => lf:   (1,(3,()))
```


Demonstrate filtering a list of 4 integers to create a new list with just the odd values. Note that we didn't modify the original list.

There are *two* folds:

foldl (left) and

foldr (right).

There are two folds because of the way they group the elements as they parse them, either grouping from the left or the right, as we'll see.

Add foldl to List

`accum` is the *accumulator*.

`f` takes two args, `accum` and each item, and returns a new `accum`.

```
def foldl(accum, &f)
  tail.foldl(
    f.call(accum, head), &f)
end
def EMPTY.foldl(accum,&f)
  accum
end
```

`tail.foldl(…)` is called *after* calling `f.call(…)`

Friday, April 12, 13

Foldl calls tail.foldl after calling f.call(accum, head). Note that it “groups” the accum with the first element, then works down the list.

Add foldr to List

f takes two args, each item and accum, and returns a new accum.

```
def foldr(accum, &f)
  f.call(head,
    tail.foldr(accum, &f))
end
def EMPTY.foldr(accum,&f)
 accum
end
```

tail.foldr(...) is called *before* calling f.call(head,...)

Friday, April 12, 13

Foldr calls tail.foldr before calling f.call(head,accum). Note that it “groups” the accum with the last element (because head isn’t handled until the whole recursion finishes!), so it works down to the end of the list first, then builds the accumulator on the way back up.
Note that the arguments to f are reversed compared to foldl. We’ll see why this is useful in a moment.

Example of foldl

```
ll = list.foldl(0) {|s,x| s+x}
lls= list.foldl("0") {|s,x|
   "(#{s}"+#{x})"
}
puts "ll:  #{ll}"
puts "lls: #{lls}"
# => ll:  10
# => lls: ((((0+1)+2)+3)+4)
```


Sum the list using foldl and also build a string that shows us who it proceeded!

Example of foldr

```
lr = list.foldr(0) {|x,s| x+s}
lrs= list.foldr("0") {|x,s|
  "(#{x}"+#{s})"
}
puts "lr:  #{lr}"
puts "lrs: #{lrs}"
# => lr:  10
# => lrs: (1+(2+(3+(4+0))))
```


Sum the list using foldr and also build a string that shows us who it proceeded! Note that the block has the x and s args reversed compared to foldl! This is conventional so the accumulator shows up in the last position, as shown in the string.

Compare foldl, foldr

```
foldl: ((((0+1)+2)+3)+4) == 10
foldr: (1+(2+(3+(4+0)))) == 10
```

The *sums* are the same,
but the *strings* are *not*!
Addition is *commutative* and *associative*.

Compare the left recursion with the right recursion. Note that reversing the block args for foldr resulted in this clearly formatted string showing the right recursion. This is why people like to use that convention.
The additions were the same because + is commutative, but the string formation isn't, as the two strings are different!

Try *subtraction*

```
foldl: (((0-1)-2)-3)-4) == -10
foldr: 1-(2-(3-(4-0)))) == -2
```

Substitute - for +.
Subtraction is *neither*
commutative nor *associative*.

If you substitute – for +, you'll also get different results.

Friday, April 12, 13

Let’s look at one of the functional data structures, List, which we’ve already looked at a bit, but we need to explore further.

filter, map and fold

as *modules*...

So, we looked at these. What's the big deal?? They are excellent examples of why functional programming is the right approach for building truly modular systems...

A Good *Module*:

`interface`	Single responsibility, clear abstraction, hides internals
`composable`	Easily combines with other modules to build up behavior
`reusable`	Can be reused in many contexts

Here are some of the qualities you expect of a good “module”. It exposes an interface that focuses on one “task”. The use of the abstraction is clear, with well defined states and transitions, and it’s easy to understand how to use it. The implementation is encapsulated.
You can compose this module with others to create more complex behaviors.
The composition implies reusability! Recall that it’s hard to reuse anything with side effects. Mutable state is also problematic if the module is shared.

Group email addresses

Exercise: implement `List.make`

```
addrs = List.make(
"Dean@GMAIL.COM",
"bob@yahoo.com",
"tom@Spammer.COM",
"pete@YAHOO.COM",
"bill@gmail.com")
```

Let's *convert* to lower case, *filter* out spammers, and *group* the users by address…

Let's group a "huge" list of email address.

Group email addresses

```
grouped = addrs.map {|x|
 x.downcase
}.filter {|x|
  x !~ /spammer.com$/
}.foldl({}) {|grps,x|
  name, addr = x.split('@')
  l = grps[addr] || List::EMPTY
  grps[addr] = List.new(name,l)
  grps
}
```


We first map each string to lower case, then remove the strings that end with “spammer.com”, using a regular expression, and finally fold over the remaining items. The fold takes an empty hash map {} as the initial value. We split each string on ‘@’, then initialize the list of names for that address, if not already initialized. Now we create a new list, adding the name, and reassign to the hash map. Finally, the block has to return the hash map for the next pass (or the end of the foldl).
Note: there is mutation of the hash map going on, but it is local to this thread!

Group email addresses

```
...
grouped.each {|key,value|
 puts "#{key}: #{value}"
}
=> yahoo.com: (pete,(bob,()))
=> gmail.com: (bill,(dean,()))
```

We calculated this grouping
in *10 lines of code*!!

For nice output, iterate over the hash map with “each” and print each key-value pair on its own line.

If we had GroupedEmailAddresses objects,

how much more *code* would be *required*?

10 lines of code, reusing filter, map, and fold vs. how much custom, one-off code?

How much more *development time* would be *required*?

Friday, April 12, 13

How much time would you spend implementing the custom solution?

filter, map, and fold are ideal *modules*.

Each has a *clear abstraction*,
composes with others,
and is *reusable*.

What makes them so modularity is their stability, clear abstraction, near infinite composability to build higher-order abstractions, which implies reusability!

filter, map, and fold are *combinators*.

The term "combinator" is a technical term in FP. For our purposes, these functions take other functions as arguments, which is how they are adapted to different purposes, and they combine with each other to build up more sophisticated "calculators".


The Haystack, Oregon
Better Objects
57


Immutable
Values
Better Objects

Immutable values are better for *concurrency* and they minimize obscure bugs because of *side effects.*

If you must do multithreaded programming, it’s far easier if your values are immutable, because there is nothing that requires synchronized access. Also, obscure bugs from “non-local” side effects are avoided.

Immutability tools

- *final* or *constant* variables.
- No field "setter" methods.
- Methods have no side effects.
  - Methods return new objects.
- (Persistent data structures.)

Friday, April 12, 13

These techniques help you achieve immutability in any language. Persistent data structures let you make "copies" of big data structures efficiently. (See my tutorial for details about them…)


TDD
Better Objects

Test Driven Development

(including *refactoring*)

is still useful in FP,

but there are *changes*.

If you must do multithreaded programming, it's far easier if your values are immutable, because there is nothing that requires synchronized access. Also, obscure bugs from "non-local" side effects are avoided.

First, you tend to use *more experimentation* in your *REPL* and *less test first*.

It’s somewhat like working out a math problem. You experiment in your Read Eval Print Loop (interactive interpreter), working out how an algorithm should go. They you commit it to code and write tests afterwards to cover all cases and provide the automated regression suite. The test-driven design process seems to fit less well, but other people may disagree!

Testing *Money*

```
class Money
 PRECISION = 0.00001
 attr_reader value
 def initialize value
  @value = round(value)
 end

 def round value
  # return rounded to ? digits
 end
 …
end
```


Money is a good domain class to implement as a “functional” type, because it has well-defined semantics and supports several algebraic operations!
The round method rounds the value to the desired PRECISION. I picked 5 decimal places, even though we normally only show at most a tenth of a penny...

Testing *Money*

```
…
def add other
  v = other.instance_of?(Money) ?
        other.value : other
  Money.new(value + v)
end
…
end
```


The add method tests the value to see if it's another Money or a (assumption) a float. It returns a new Money (of course!)

Imaginary *RSpec*

```
describe "Money addition" do
 money_gen = Generator.new do
  Money(-100.0) to Money(100.0)
 end
...
```

Define a “generator” that generates a random sample of instances between the ranges shown.

RSpec is a popular Ruby testing framework in the style of Behavior Driven Development (BDD). I am showing fictitious extensions to illustrate a particular functional approach – testing properties that should hold for all instances. So it’s less about “testing by example” and (as much as is possible) testing universal properties.

We start by defining a function that can generate N random sample instances within an arbitrary range.

Imaginary *RSpec*

```
describe "Money addition" do
 money_gen = Generator.new do
  Money(-100.0) to Money(100.0)
 end
 property "is commutative" do
  money_gen.make_pairs do |m1,m2|
   m1.add(m2).should_be_close(
   m2.add(m1), Money::PRECISION)
  end
 end
end
```

verify that addition is commutative!

In our fictitious RSpec extensions, we verify the property that addition is commutative. We ask the “money_gen” to create some random set of pairs, passed to the block, and we verify that m1+m2 == m2+m1 within the allowed precision.

Test Driven Development

becomes

property verification.

Of course, you’ll still write a lot of conventional OO-style tests, too.

Refactoring?

```
grouped = addrs.map {|x|
 x.downcase
}.filter {|x|
  x !~ /spammer.com$/
}.foldl({}) {|grps,x|
  name, addr = x.split('@')
  l = grps[addr] || List::EMPTY
  grps[addr] = List.new(name,l)
  grps
}
```

How might you *refactor* this code?

Friday, April 12, 13

Remember how we grouped email addresses? What is refactoring like in such a world?

Recall

```
grouped = addrs.map {|x|
  x.downcase
}.filter {|x|
  x !~ /spammer.com$/
}.foldl({}) {|grps,x|
  name, addr = x.split('@')
  l = grps[addr] || List::EMPTY
  grps[addr] = List.new(name,l)
  grps
}
```

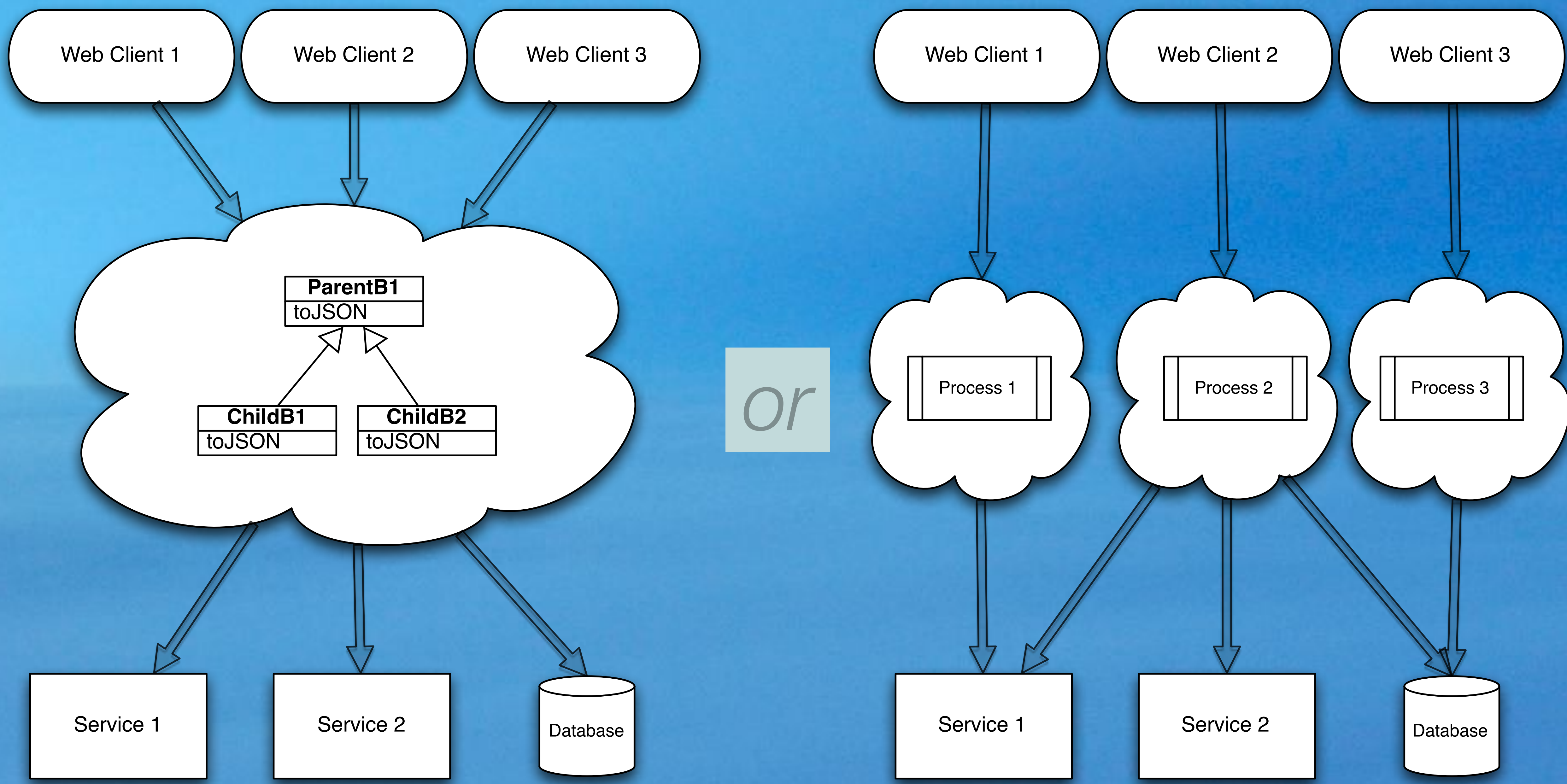
Extract Function?

We could extract some of these blocks into Ruby “procs” that we pass in to the methods. This would make the code less dense and provide opportunities for generalization (e.g., pluggable spam address filters).
We can also do traditional refactoring of some of the lines in the foldl block. However, let’s avoid premature refactoring! If the extracted function is never used anywhere else, don’t extract it, unless clarity is a problem.


OO
Middleware
Better Objects
71

In a *highly-concurrent* world, do we really want a *middle*?

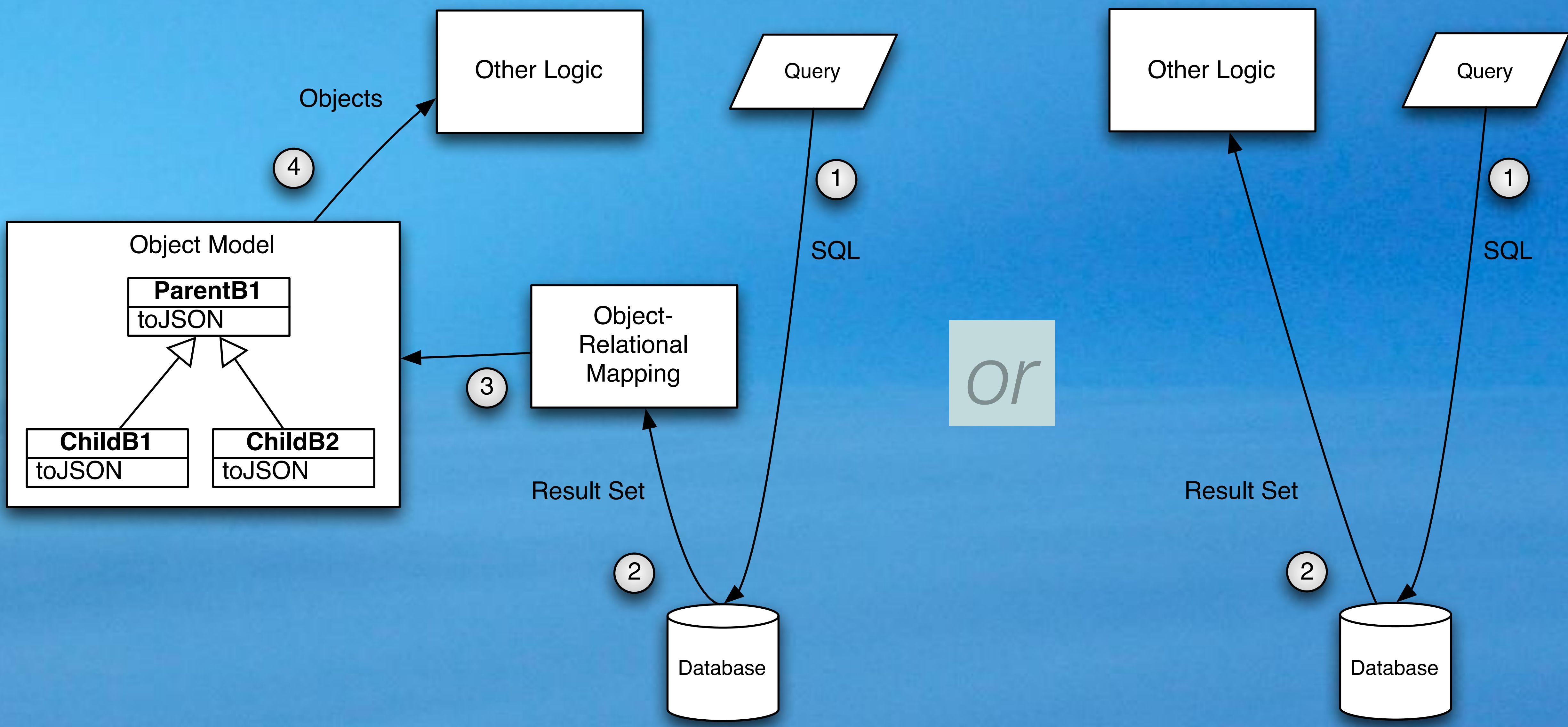
Which *Scales* Better?



Friday, April 12, 13

If we funnel everything through a faithfully-reproduced domain object model, our services will be bigger, harder to decompose into smaller pieces, and less scalable. *Modeling* our domain to understand it is one thing, but implementing it in code needs to be rethought. The compelling power of combinators and functional data structures are about as efficient and composable as possible. It's easier to compose focused, stateless services that way and scale horizontally.

What about *ORM*?



Question Object-Relational Mapping

What if your business logic just worked with the collections returned from your database driver? It’s true that some of these collections, like Java’s ResultSet, don’t have the powerful combinators we’ve been discussing, but those “methods” could be added as static service methods in a helper class.
The question to ask is this: does the development and runtime overhead of converting to and from objects justify the benefits?

Object *middleware*, including *ORM*, isn't *bad*. It just has *costs* like everything else...

Just remember that every design decision has costs, so evaluate those costs with a clear head...

Friday, April 12, 13
(Nehalem State Park, Oregon)

Concurrency is the reason people started discussing FP, which had been primarily an academic area of interest. FP has useful principles that make concurrency more robust and easier to write.

(San Francisco Bay)

We're Drowning in Data.

twitter

facebook

YouTube

...

Not just these big companies, but many organizations have lots of data they want to analyze and exploit.

(San Francisco)

I will argue that objects haven't been the modularity success story we expected 20 years ago, especially in terms of reuse.

(Mud near Death Hollow in Utah.)

We need better agility.

Schedules keep getting shorter. The Internet weeded out a lot of process waste, lot Big Documents Up Front, UML design, etc. From that emerged XP and other forms of Agile. But schedules and turnaround times continue to get shorter.

(Ascending the steel cable ladder up the back side of Half Dome, Yosemite National Park)

Every now and then, we need to stop, look at what we're doing, and remove the cruft we've accumulated. I claim that a lot of the code we write, specifically lots of object middleware, is cruft.

(Maligne Lake, Near Jasper National Park, Jasper, Alberta)

Thank You!

- dean@deanwampler.com
- @deanwampler
- polyglotprogramming.com/talks