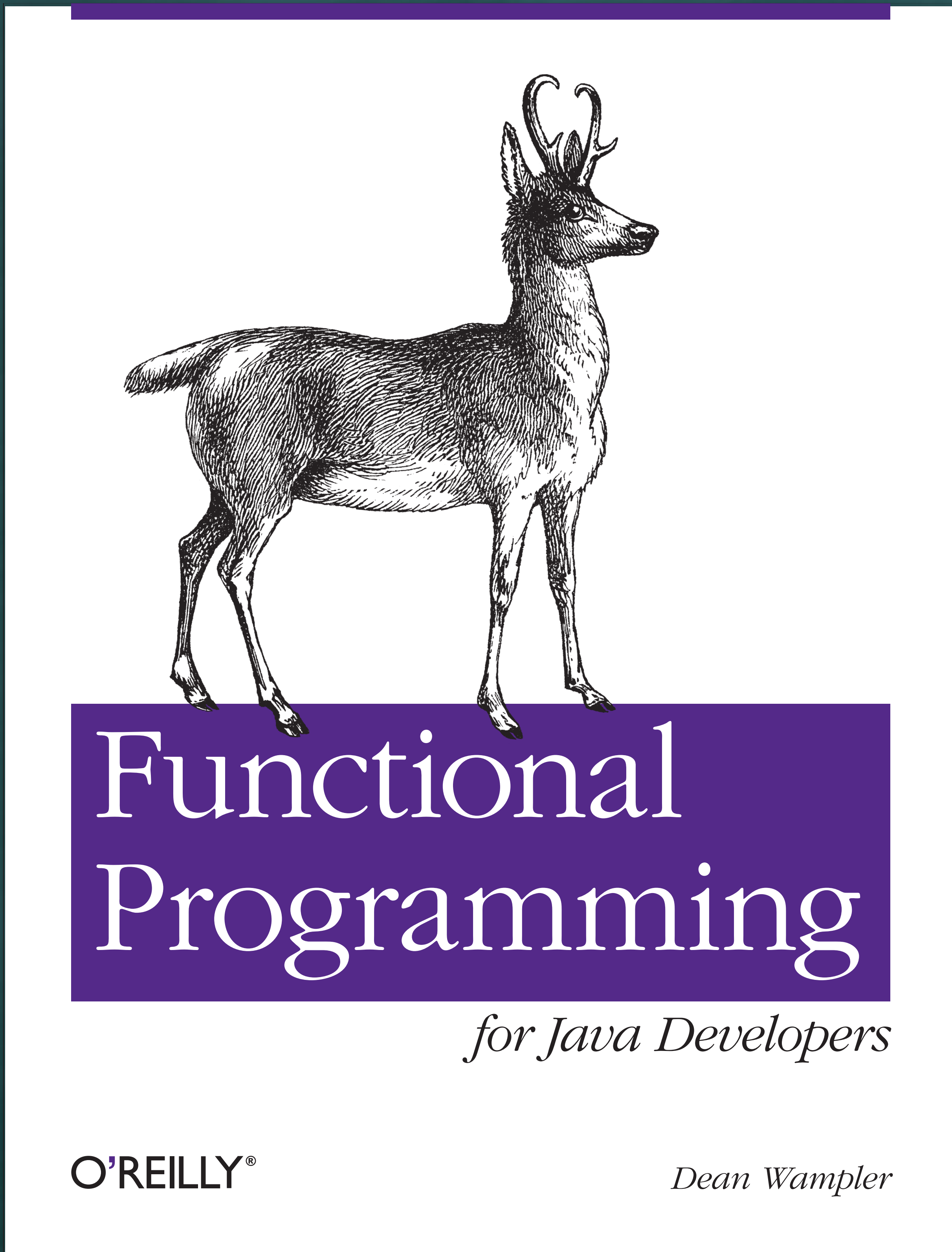
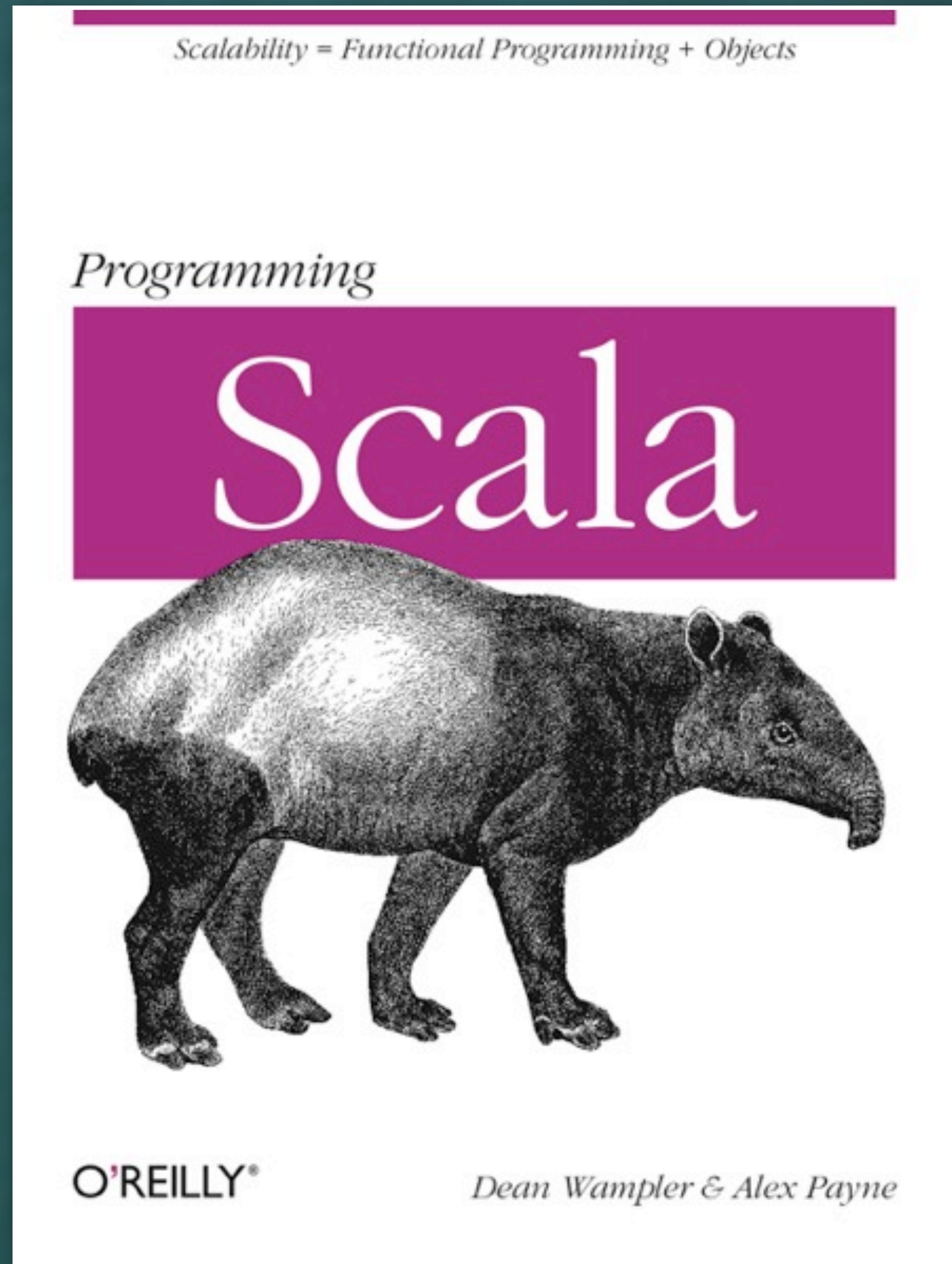
Friday, April 12, 13

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, taken before last year's OSCON. Some are from the San Francisco area, including the Bay. A few are from other places I've visited over the years.

(The Haystack, Cannon Beach, Oregon)



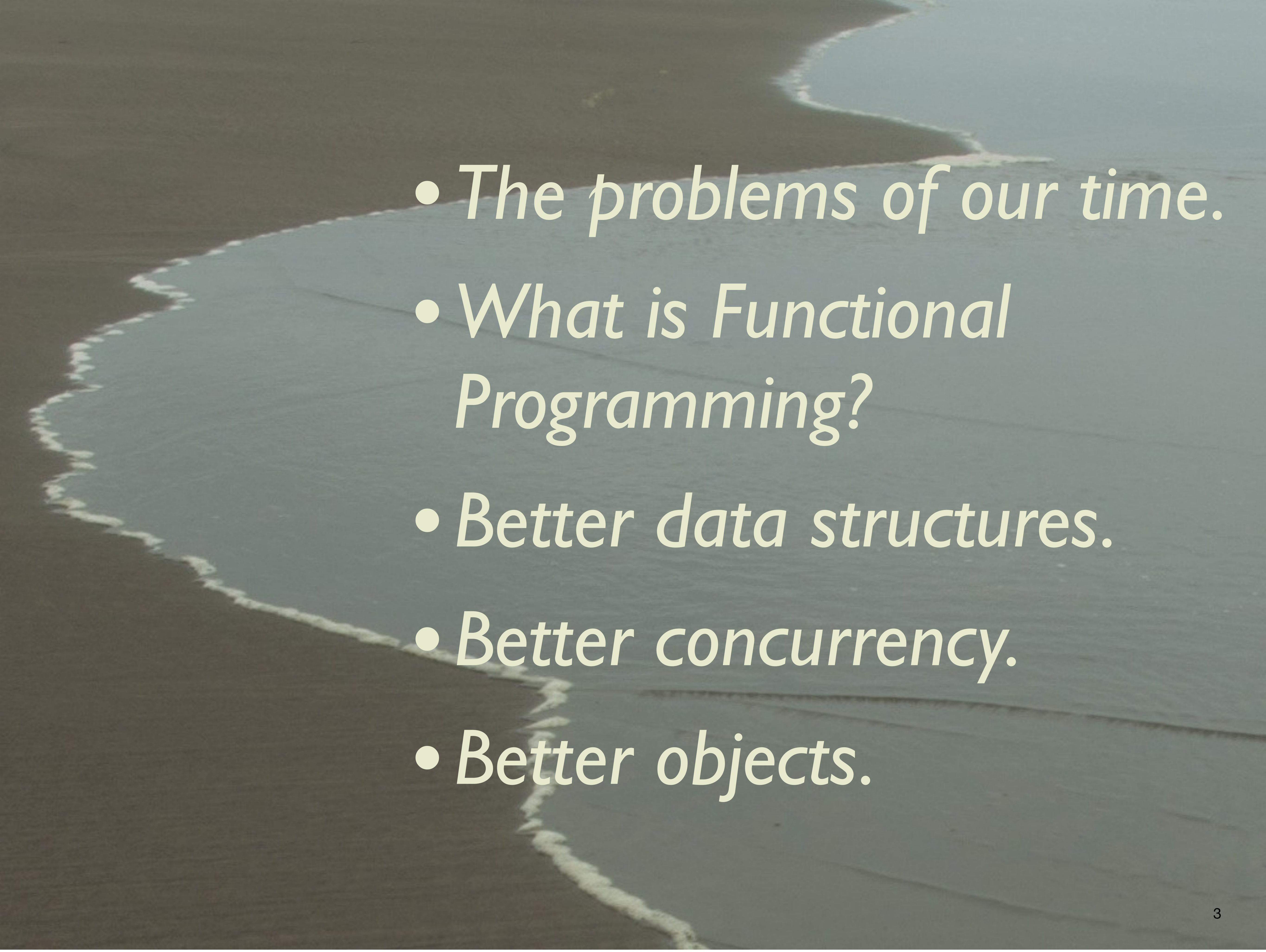


polyglotprogramming.com/fpjava

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.

(Nehalem State Park, Oregon)

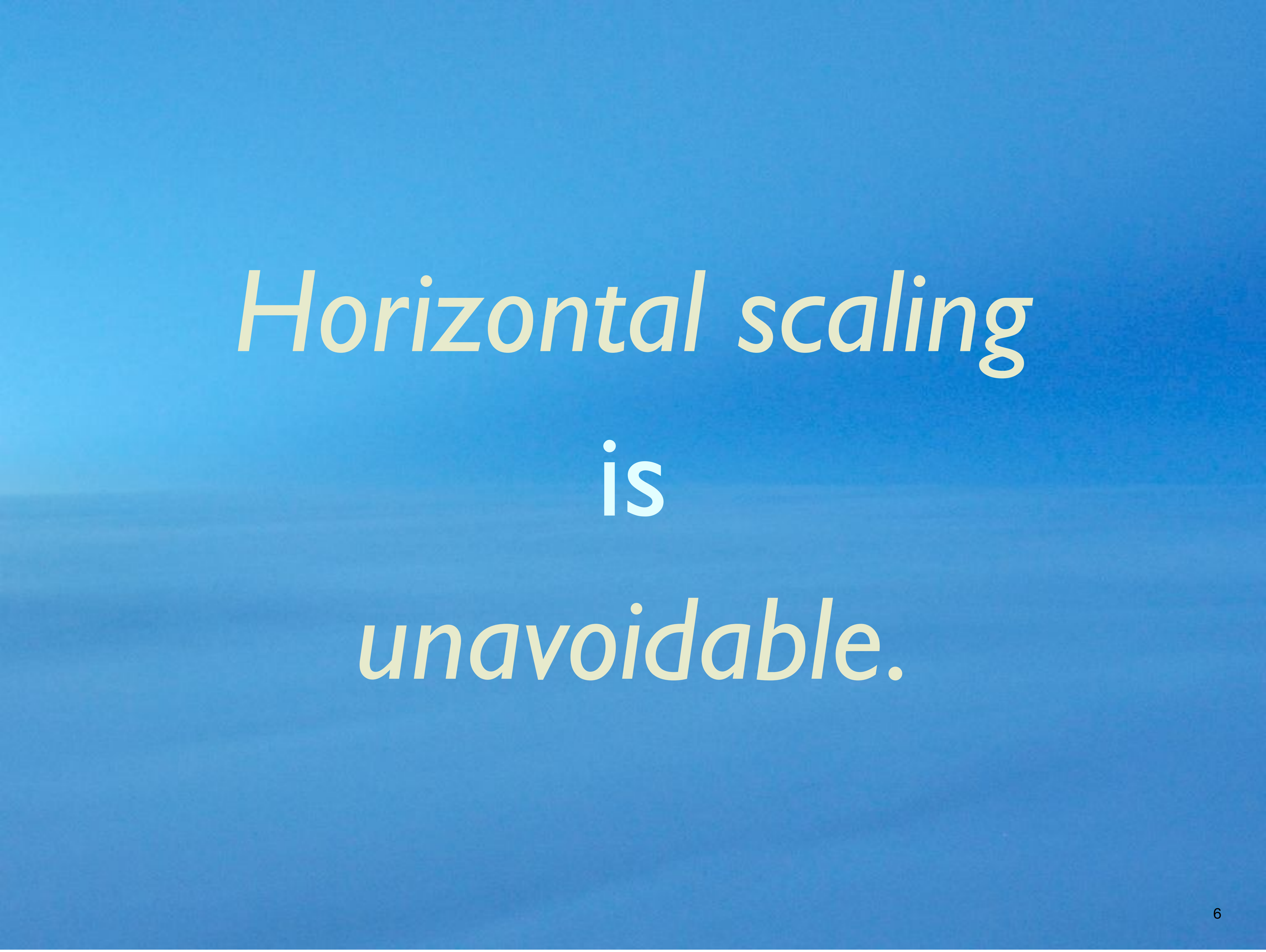
Friday, April 12, 13

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)



Friday, April 12, 13

The reason everyone is talking about concurrency is because we've hit the limit of vertical scalability of Moore's Law. Now we're scaling horizontally, so we need to know how to exploit multiple CPUs and cores.

(At dusk flying over the Midwest - lightened)

Multithreaded programming

is the

assembly language

of concurrency.

Multithreaded programming is hard and too low-level for most needs.

We’re Drowning in Data.

...

Friday, April 12, 13

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.

Friday, April 12, 13

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)

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...

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.

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?

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.

Java

```
class List<T> {
  final T       _head;
  final List<T> _tail;
  T       head() {return _head;}
  List<T> tail() {return _tail;}

  List (T head, List<T> tail) {
   _head = head; _tail = tail;
  }
  …
}
```


I'll provide some Java examples, but mostly Ruby examples, since its syntax is compact and relatively easy to learn - both good for presentations like this!
Here's a linked list that we'll use a lot. It is defined by the head of the list (the left-most element) and the tail or rest of the list, itself a list! Make the fields final in Java and don't provide setters. (I'm dropping public, private, etc. for clarity.) List objects will be immutable, although we can't control the mutability of T objects!
If you don't like static typing, at least appreciate the fact that you know immediately that tail is also a List<T>.
I'm not using JavaBeans conventions here to reduce unnecessary clutter. In fact, is there any reason to NOT make the fields public?

Java

```
List<? extends Object> list =
  new List<Integer>(1,
    new List<Integer>(2,
      new List<Integer>(3, …)));
```


Friday, April 12, 13

Creating a list (we'll see less verbose syntax later).
I'm showing *covariant typing*, a poorly understood feature in Java (and it could be implemented better by the language…).
Read this as, "I declared list to be of List<T> for any subtype T of Object, so List<String> is a subtype of List<Object>, and a valid object to assign to list." NOTE: this is *different* than assigning Integers (and Strings and Floats and…) to a List<Object>.
How should we terminate this list?? What should the final tail be?? We'll come back to that.

Ruby

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


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, …)))
```


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

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

Friday, April 12, 13

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.

Side-effect free methods and immutable objects

```
class List
 …
 def add(item)
  List.new(item, self)
 end
 …
end
```

Make your *methods* side-effect free.
Create *new* instances.

Don't modify the existing list, make a new one.
(We won't have time discuss how you optimize making copies to minimize that overhead…)


What is Functional Programming?
First-class functions
28

First Class Functions

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

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.

First Class Functions

```
f = lambda { |x|
  puts "Hello, #{x}!"
}
def usearg(arg, &func)
  func.call(arg)
end

usearg("Dean", &f)
# "Hello, Dean!"
```


There are other syntaxes for defining and calling Ruby procs/lambda.

First Class Functions

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

This is one of the most powerful concepts in programming.

Java?

```
public interface
Function1Void<A> {
 void call(A arg); // arbitrary
}

public static void usearg(
 String arg,
 Function1Void<String> func) {
  func.call(arg);
 }}
```


Java doesn’t have first-class functions. The closest we can come are function “objects”. Often these interfaces are instantiated as anonymous inner classes.
I picked an arbitrary name for the function.

Java?

```
public static void main(…) {
 usearg("Dean",
  new Function1Void<String>(){
   public void call(String s){
    System.out.printf(
     "Hello, %s!\n", s);
   }
  });
}
```


Verbose, ugly and hard to follow.
The ability to communicate ideas in concise terms really matters!! Your brain expends a lot of effort parsing all this code!

Java?

```
public interface
Function1Void<A> {
 void call(A arg);
}
…
public interface
Function2<A1,A2,R> {
 R call(A1 arg1, A2 arg2);
}
…
```

Another example function.

How many *one-off interfaces* could you replace with *uniform abstractions* like these?

Java APIs must have hundreds of *structurally* identical interfaces, each with its own ad-hoc interface and method name. Imagine how much memorization reduction would be facilitated if they were all replaced with uniform abstractions like these?

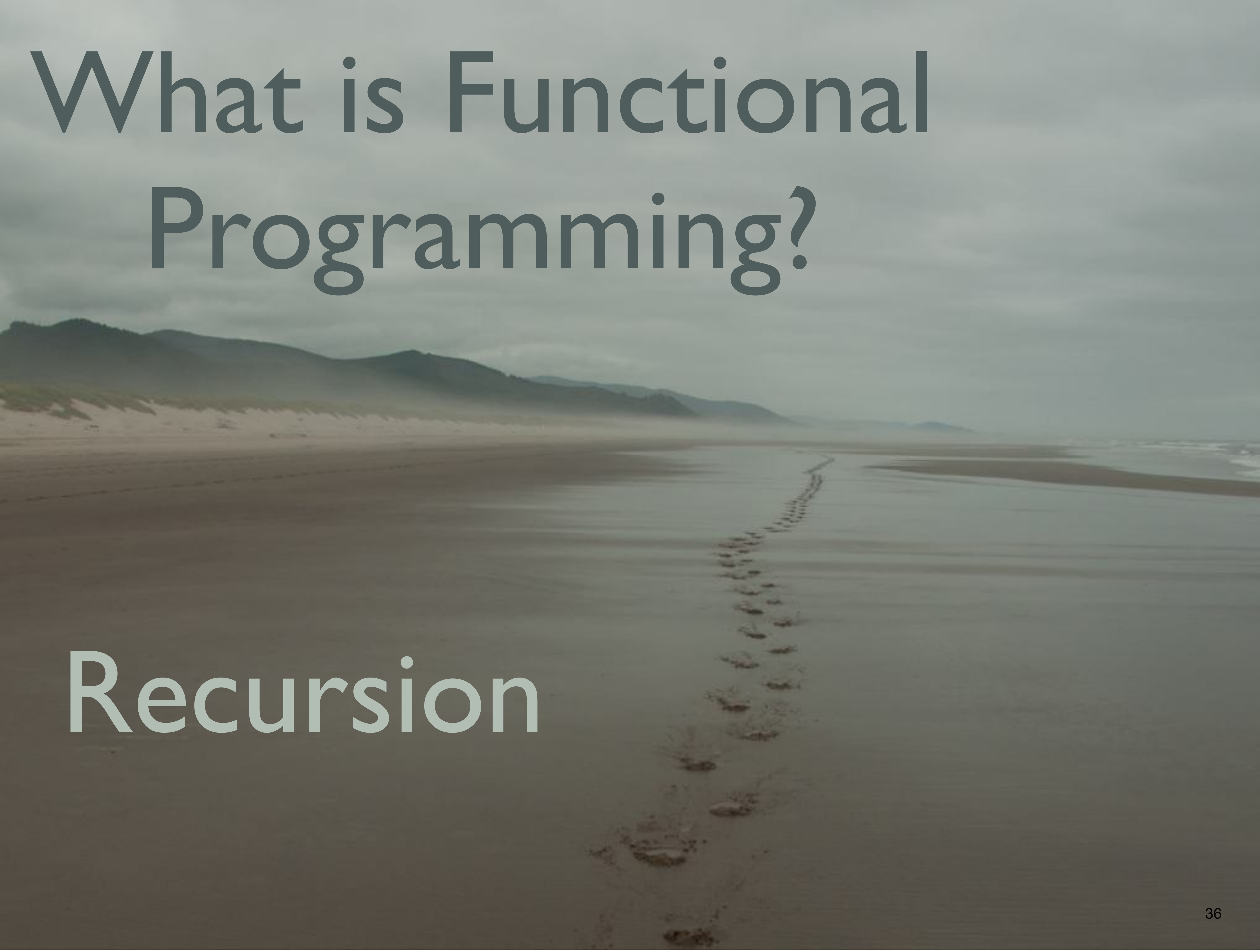
Side note: Java 8 will *hopefully*, *finally* add a lambda syntax to eliminate lots of this boilerplate.

Higher-order Functions

```
def usearg(arg, &func)
 func.call(arg)
end
```

Functions that take other functions as arguments or return them as results are called *higher-order* functions.

There's a technical reason for the name "higher-order" that we won't have time to discuss, but since you'll hear this term used, I wanted to define it in the way people typically use the term.


What is Functional Programming?
Recursion

Recursion

```
class List
  …
  def empty?
    false        # always??
  end
  def to_s
    empty? ?
      "()" :
      "(#{head.to_s},#{tail.to_s})"
  end
  …
end
```

`tail.to_s` is a *recursive* call.

Recursion is a natural tool for working with “recursive” data structures, like List. It’s also a way to traverse data structures without mutable loop counters!
Note that we haven’t shown how to represent an empty list! We will.
If the list is empty, we terminate the recursion, returning the string “()”. Otherwise, we form a string by calling head.to_s and tail.to_s. The latter is a recursive call. (We could have left off the “to_s” here, but to make things explicit...

Recursion

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

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

We'll define EMPTY shortly…

We'll define EMPTY shortly, which will have an empty? method that returns false.
If we run this code, we get the string shown. Note the nesting of parentheses, reflecting the nesting of structure!

Friday, April 12, 13

(Nehalem State Park, Oregon)

Friday, April 12, 13

(Nehalem State Park, Oregon)

Nulls are a serious source of *bugs*.

We know that nulls are a pain...

If *values* are *immutable*, can we avoid using *nulls*?

Functional programming emphasizes rigor and always having valid values assigned to variables. Can we eliminate the use of nulls?

What *should* happen?

```
Map<String,String> capitals = …;
…
String cap =
   capitals.get("Camelstan");
String cap2 = cap.toLowerCase();
```

↑

`NullPointerException`!!

`cap` is of *type* `String` *or* `Null`?

or is `Null` a *subtype* of `String`?

Friday, April 12, 13

Let’s return to Java, because this section makes more sense for statically-typed languages.
We have a map of the capital cities for the world’s countries. We ask for the capital of Camelstan, then try to use the value.
We forgot to check for null.
If null were of type Null, then could tail be thought of as a variable of type String OR Null?
Actually, Java *effectively* has the notion of a Null type that is a subtype of all other (reference) types, but not explicitly.

What *should* happen?

```
String cap =
  capitals.get("Camelstan");
```

`Map<K,V>.get` signature:

```
  V get(Object key);
```

It's *lying* slightly, because
a `V` *or* a `null` is returned.

The signature for Map.get doesn't tell the full story of what might happen. We're used to the fact that null might be returned, but naively, reading this signature, we have every right to believe a valid string will *always* be returned.

What *should* happen?

What if we changed the signature?

```
  Option<V> get(Object key);
...
Option<String> cap =
  capitals.get("Camelstan");
```

Explicitly indicate that a value might exist *or not*; it is *optional*.

The signature for Map.get doesn't tell the full story of what might happen. We're used to the fact that null might be returned, but naively, reading this signature, we have every right to believe a valid string will *always* be returned.
Type safety will prevent us from "forgetting" that the value is optional, in the same way that we can forget that null is returned.
We have to handle the option explicitly.

Option

```
interface Option<T> {
  boolean hasValue();
  T get();
}

final class Some<T> extends
Option<T> {
  boolean hasValue(){return true;}
  T get() {return t;}
  private T t;
  // constructor...
}
```


Here is the Option interface and the first of TWO implementing classes, which is why it's declared final. Some is the object instantiated when there IS a value.
Some people hate "final" because it's seen as bad for testing (you can't replace the object with a test-oriented subclass). There's no need to EVER do that here, and maintaining type safety (at least as much as we can) is more important.

Option

```
interface Option<T> {
 boolean hasValue();
 T get();
}

final class None<T> extends
Option<T> {
 boolean hasValue(){return false;}
 T get() {throw new Exception(…);}
}
```


Here is the Option interface and the second of the TWO implementing classes. None is the object instantiated when there ISN'T a value.

An *optional* value

```
Map<String,String> capitals = …;
Option<String> cap =
  capitals.get("Camelstan");
if (cap.hasValue()) {
 String cap2 =
  cap.get().toLowerCase();
  …
} else {
 logError("Camelstan …");
}
```


This code may be a little verbose, but it’s not much different than the normal null checks you’re supposed to do. Also, there are mechanisms that can be used, like providing iteration over this “collection”, that can eliminate the explicit hasValue check in many cases. For example, if you don’t care that there is no value; you’re just processing a bunch of things, some with values, some without, then you can easily ignore the without cases...

Replace *Nulls* with *Options*.

Change your APIs to use Options whenever Nulls are possible, either as return values or as optional argument values!

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.

Let's finish List

Previously...

```
class List
  ...
  def empty?; false; end
  def to_s
    empty? ?
      "()" :
      "(#{head},#{tail})"
  end
  ...
end
```


Let's finish the implementation of List. In particular, let's figure out how to terminate the list, which means representing an empty list for the tail.
I changed empty? to be all on one line, compared to the previously shown implementation.
I removed the explicit calls to to_s on head and tail in self.to_s; they will be called implicitly.

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
  raise "EMPTY list has no tail!!"
 end
 def EMPTY.empty?; true; 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 and tail methods for this object (so called “singleton methods”) to raise exceptions. We also define empty? to return true and 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 return itself!

Rewrite to_s.

```
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!*

Friday, April 12, 13

The check for empty is gone in to_s! It's not an infinite recursion, though, because all lists end with EMPTY, which will terminate the recursion.
We've replaced a condition test with structure, which is actually a classic OO thing to do.

Recall...

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

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

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

Friday, April 12, 13

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.

List is an

Algebraic Data Type.

Friday, April 12, 13

The name comes from Category Theory, which we won't get into. The key thing to note is that this is a constrained type hierarchy. There are only two allowed subtypes that can implement the abstraction. (Since this is Ruby, we didn't define an "interface" with the key methods.)

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.

Filter, map, fold

	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.

Friday, April 12, 13

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)
```


Friday, April 12, 13

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))))
```


Friday, April 12, 13

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*

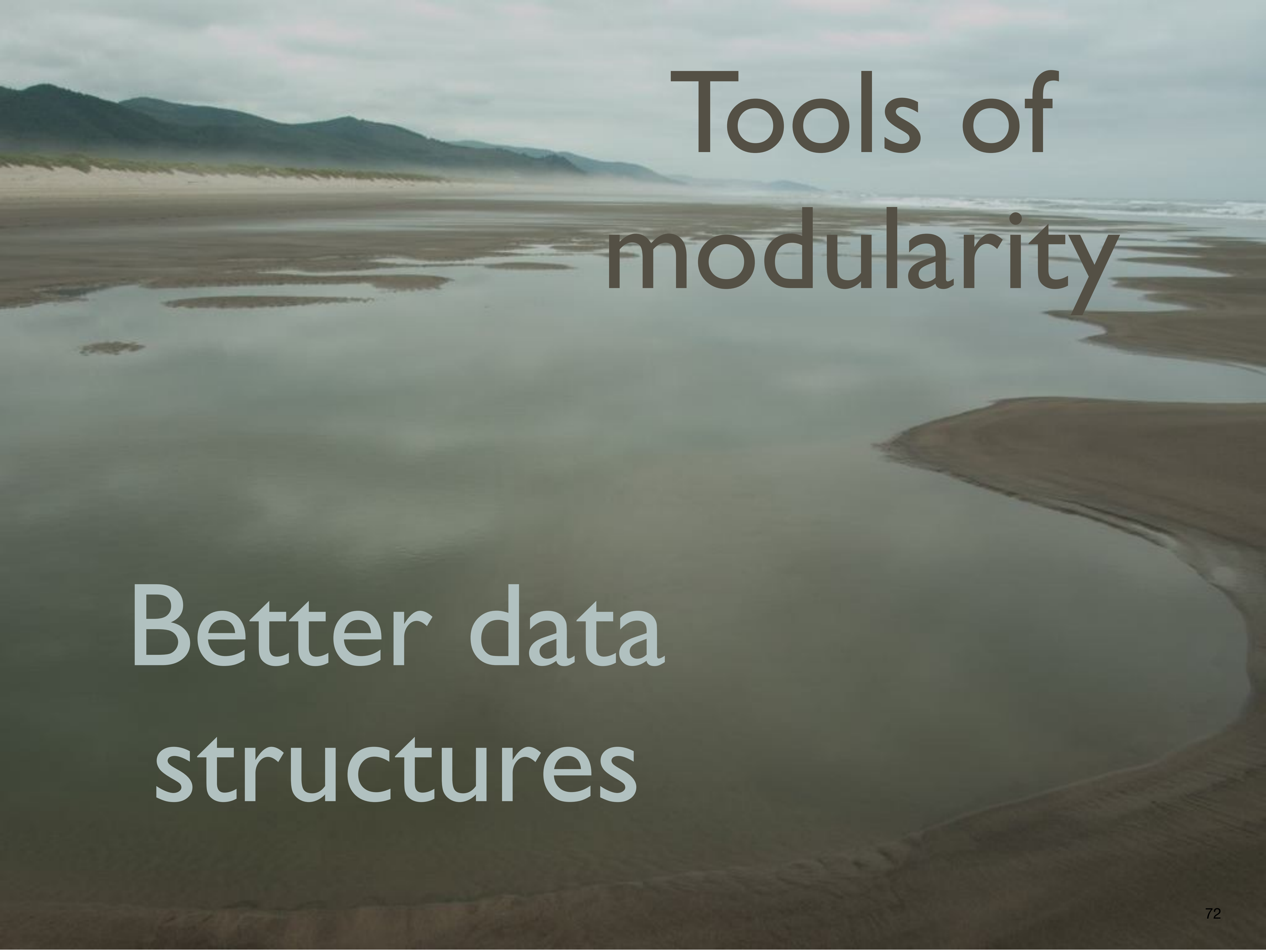
```
foldl: (((0-1)-2)-3)-4) == -10
foldr: 1-(2-(3-(4-0))) == -2
```

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

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

foldl and foldr yield *different* results for *non-commutative* and *non-associative* operations.

There are other differences that we won't have time to discuss, but you might explore. foldl is tail-recursive, so it can be optimized into a loop (if your language or VM supports that - many don't). In contrast, foldr isn't tail-recursive, but it can be used to fold over infinite data structures when only a finite subset of it is used.



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*.

Friday, April 12, 13

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”.

Aside:

Did we just break the *Law of Demeter*?

```
addrs.map{…}
     .filter{…}
     .foldl(…){…}
```


LoD says it's bad to chain method calls together, because each "link" introduces new object dependencies into the code and every time the signature for one of these methods changes, it breaks this code. It's a small that indicates the calculation should be moved to a more appropriate place.
That's not an issue here. First, we keep returning a List (except at the end), so we aren't adding dependencies. Second, map, filter and fold are so stable, they are unlikely to ever change.

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.

Isn’t copying *immutable* values *inefficient.*

We said earlier that values should be immutable, but what if they are huge? Isn’t it too expensive to make copies?

Structure Sharing

Recall...

```
class List
  def prepend(head2)
    List.new(head2, self)
  end
  ...
end
```

Note: we're *sharing* the original list with the new list: *Structure Sharing*

Friday, April 12, 13

If we prepend an element to an existing list, we create a new list, but share the old list as the tail. Recall that this would not be safe if lists or their elements are mutable.

Structure Sharing lets us "copy" values *efficiently*.

But it only works if the objects are *immutable!!*

What about Maps, Sets, Vectors, ... ?

Separate the *abstraction* from the *implementation*...

List easily supports structural sharing, what about other data structures? If we separate the external interface from the internal implementation, we can implement these types with data structures that provide efficient copies, also using structure sharing.

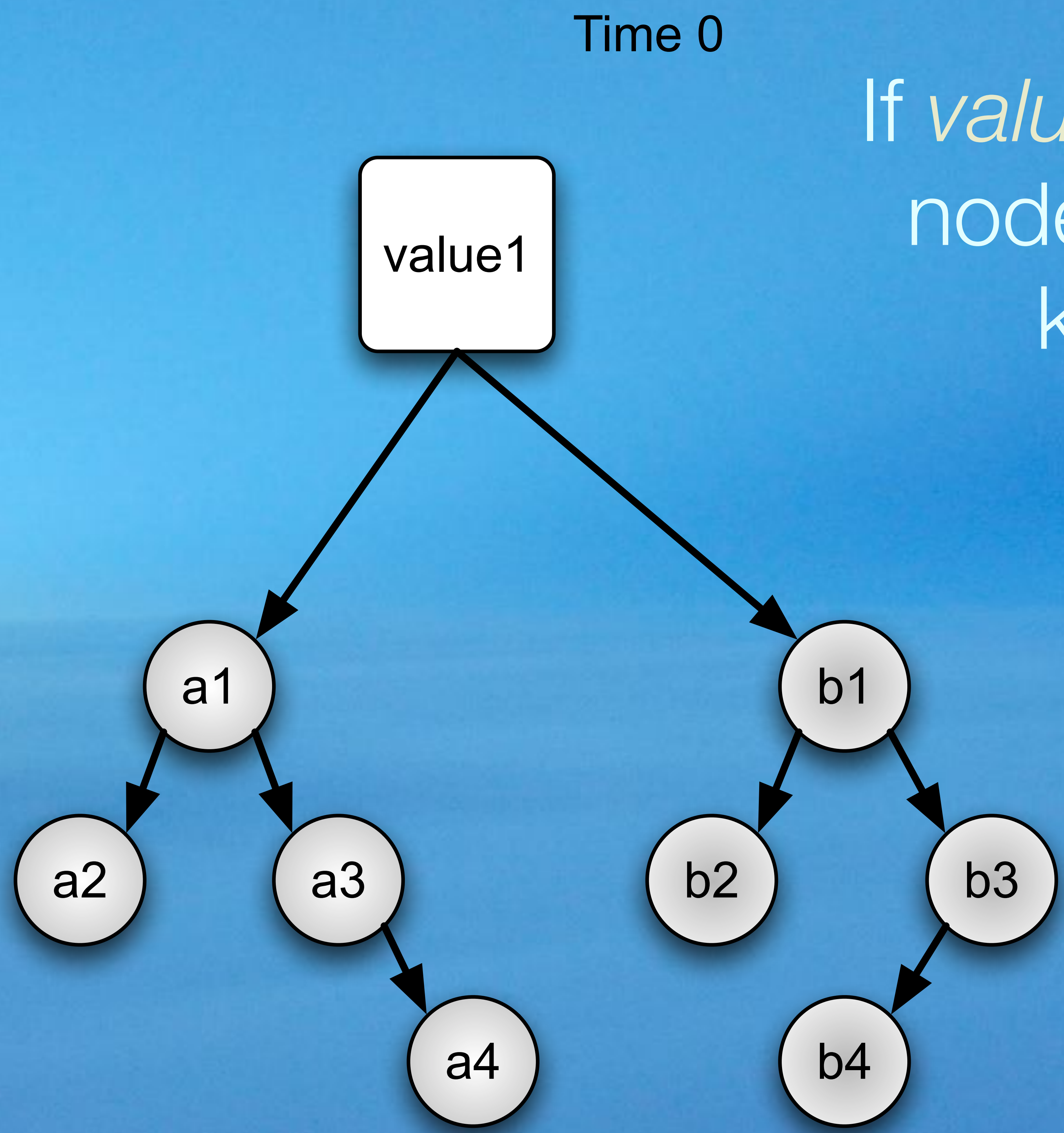
Trees enable structure sharing and provide *O(log(n))* access patterns.

The choice of tree data structure is important, but we won’t get into that.

For simplicity, we'll just use unbalanced binary trees: average *O(ln(n))*.

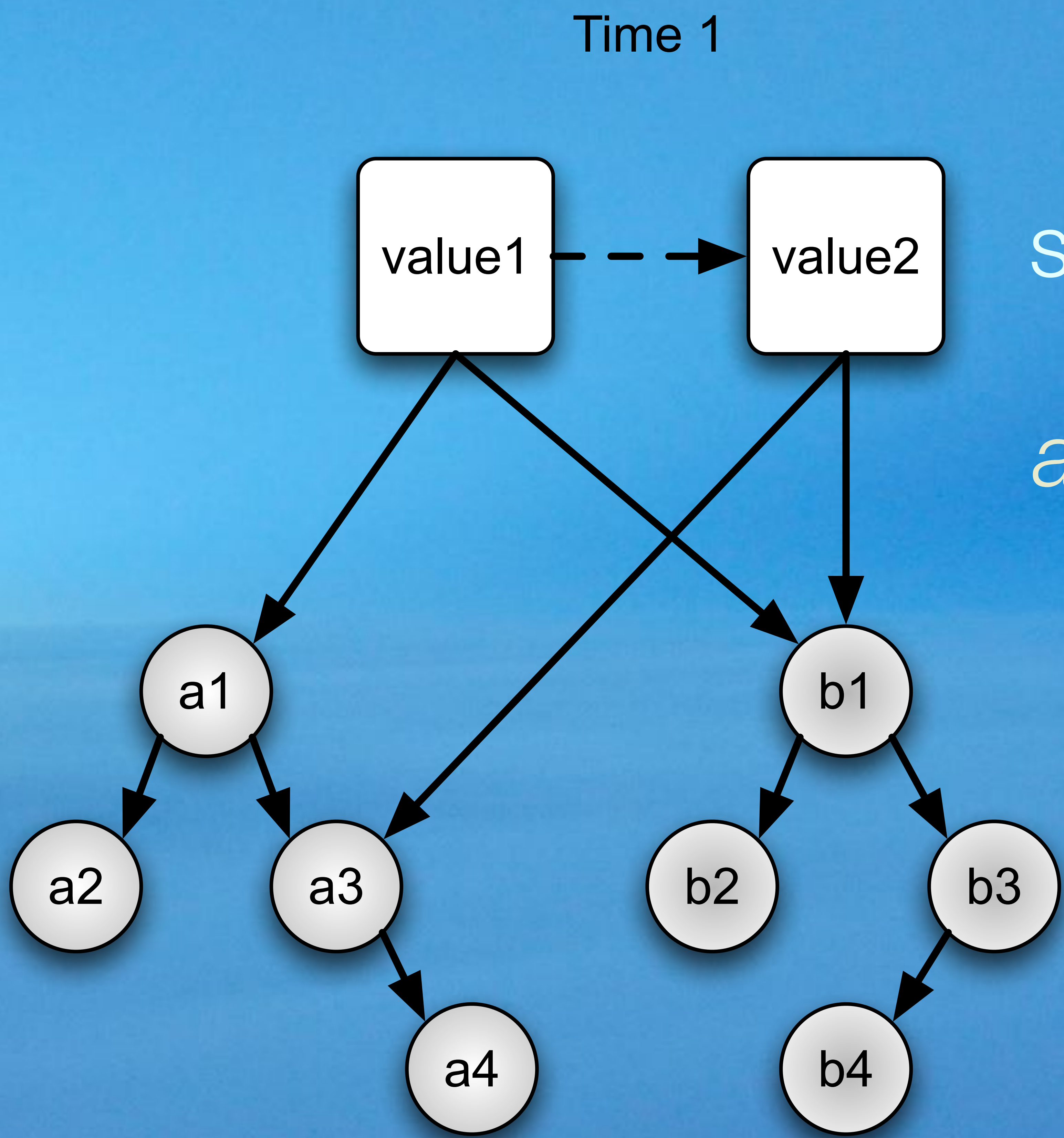
Friday, April 12, 13

Binary trees average order log base 2(n) performance. Unbalanced is generally bad, but ignoring balancing lets us focus on the key concept, structure sharing. Real implementations might use 32-way trees, giving log base 32(n) performance, and use one or another balanced tree types. The choice of implementation is made to optimize search, insertion, cache locality in the CPU, or other performance goals.



If *value1* is a Map, each node might contain a key-value pair.

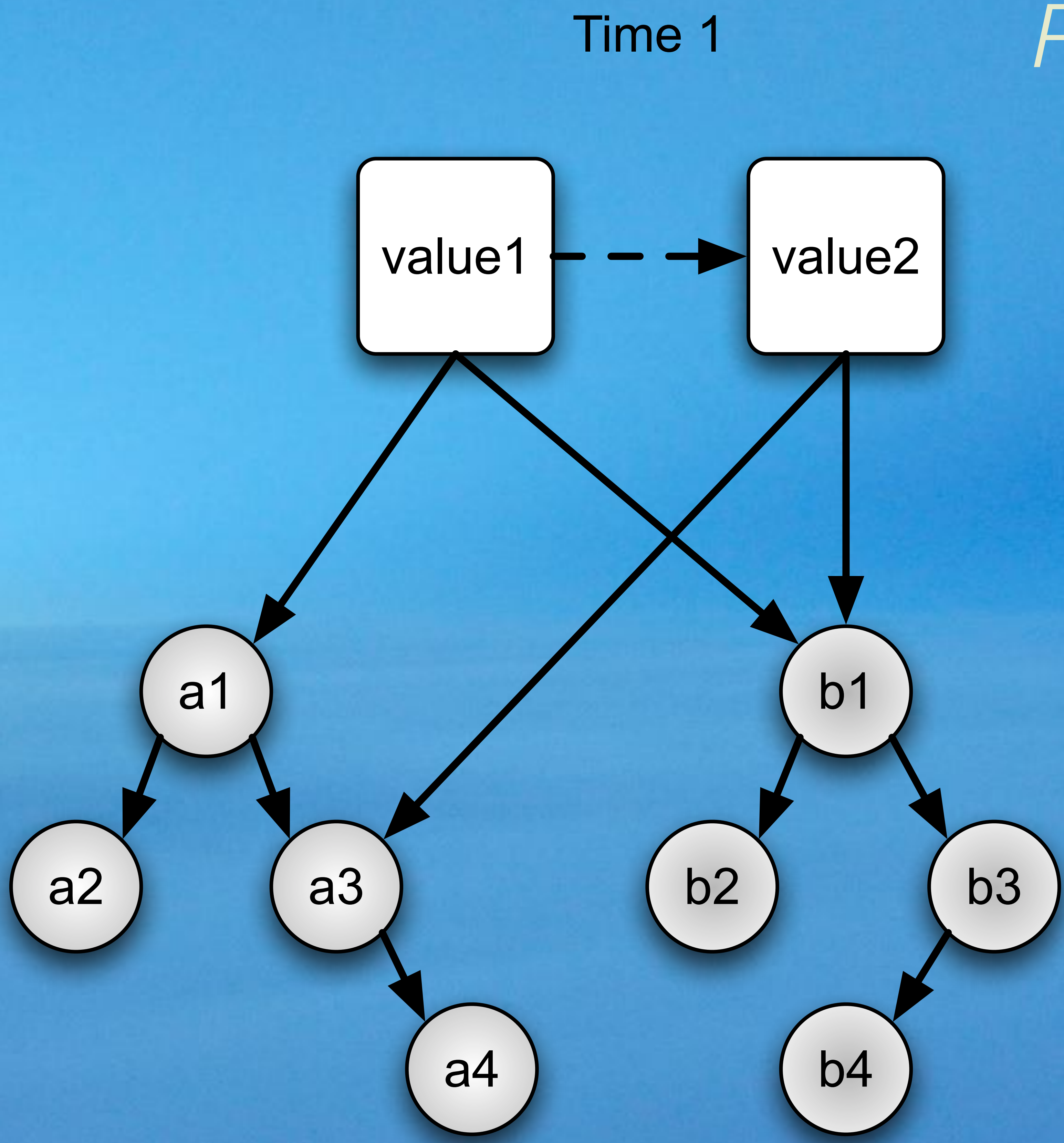
Consider value1, which implements a Map, where the



At time 1, *value1* still exists. The new *value2* reuses *a3*, *a4*, and the *b1* tree

Friday, April 12, 13

Consider at some later time, Time1, a new value2 is created that "mutates" value1, but in fact, it just introduces a new root node and shares much of value1, which still exists!



Persistent because previous values *still exist*.

Friday, April 12, 13

So, a “history” of versions is maintained, as long as there are references to the old versions. It’s not persistent in the database sense (although you could store these to disk…).

Better Concurrency

Better Concurrency

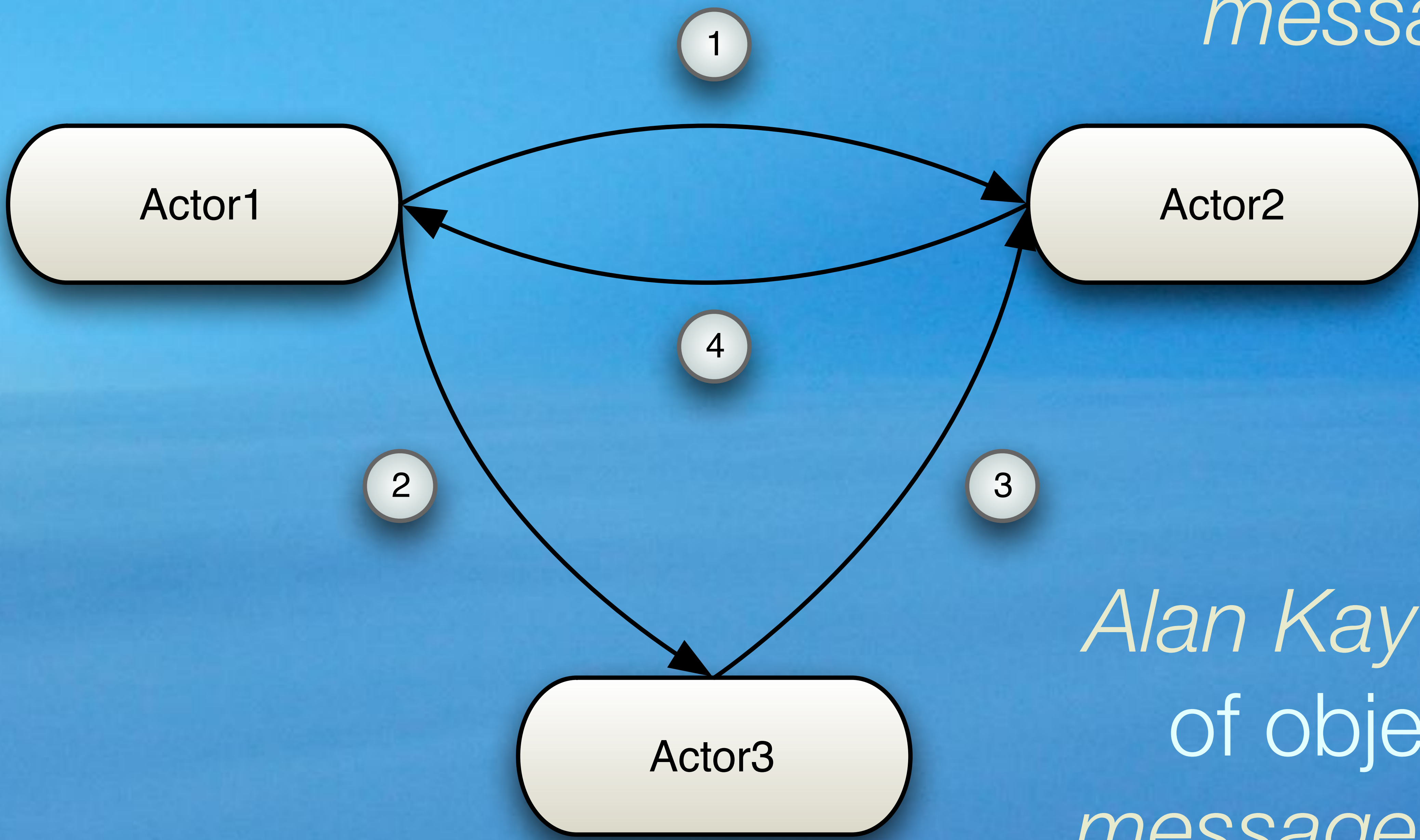
Actors

The Actor Model of Concurrency

is not specifically functional, but it follows the theme of *principled* mutation.

This is not a model that came out of the functional research community, but it fits the principle of finding “principled” ways to handle and control mutation.

Actors coordinate work by sending *messages*.

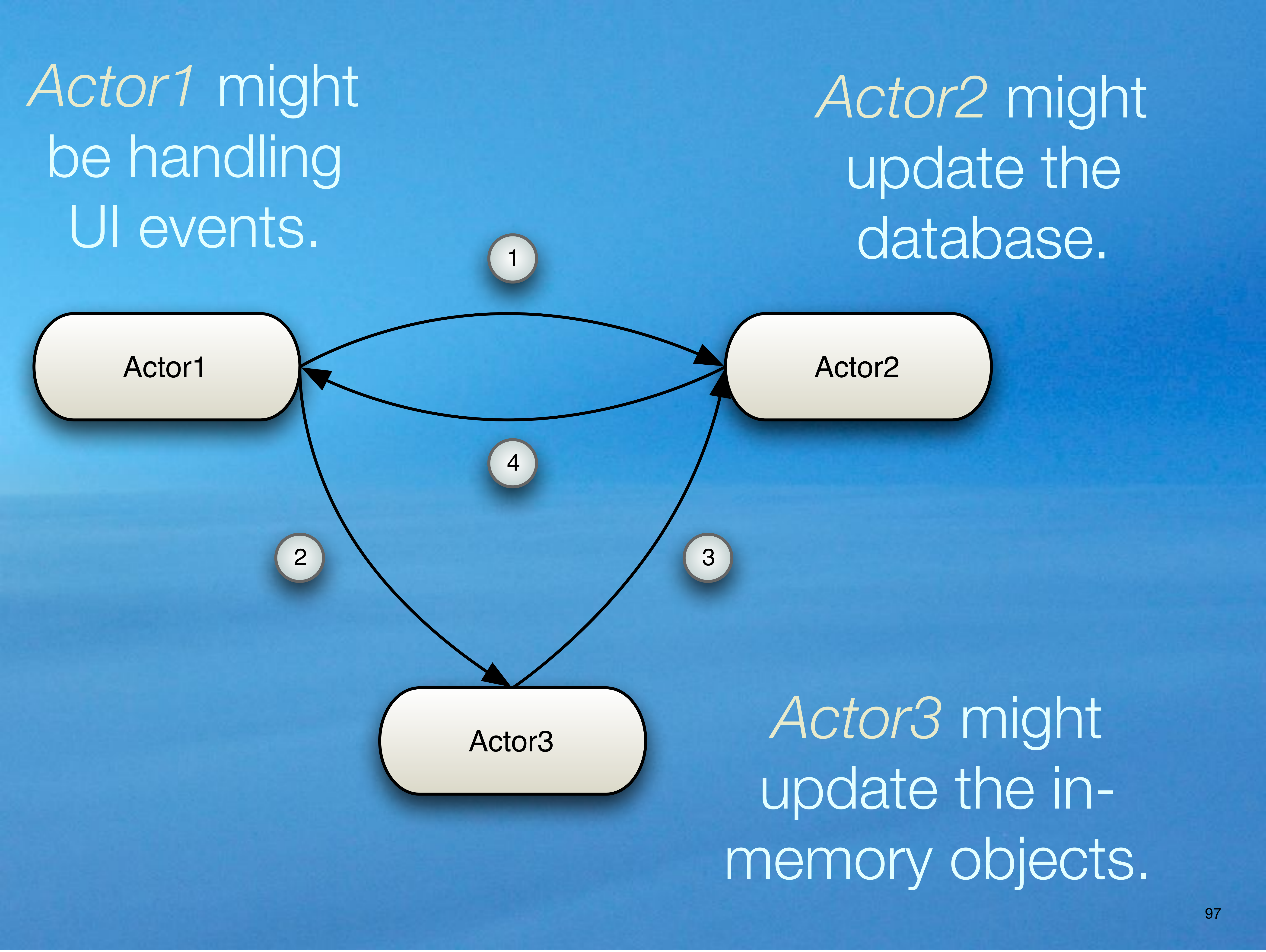


Alan Kay thought of objects as *message-passing* entities.

A schematic view. Each actor sends a message, which resides in the receiver's mailbox to be processed one at a time. When finished, the receiver can send a reply, send a message to a different actor, or do nothing.

Alan Kay, the inventor of Smalltalk, had this model in mind (although not in name) as his vision for objects; message passing entities that coordinate state mutation this way!

Erlang recently made the actor model "famous" (It was invented in the '70s by Hewitt and others).



Friday, April 12, 13

Each actor has specific responsibilities and “owns” specific mutable values. This is one model of using them, but be careful about bottlenecks!

Actor Libraries

Java	Akka, FunctionalJava, Kilim
Ruby	Revactor, Omnibus, Akka through JRuby!
. . .	Your language probably has an Actor library, too.

Google for the actor libraries for your language.

Akka Example

```
import akka.actor.*;
import static akka.actor.Actors.*;
import java.util.*;

public class MemoryActor
    extends UntypedActor {
  final Map<String,Date> seen =
   new HashMap<String,Date>();

  public void onReceive(…) {…}
}
```


Let’s see a Java example using Akka’s Actors. Note that you could do this with JRuby, too!
We declare an actor that will “remember” the messages (treated as strings for simplicity) that it receives, along with the times they were received. We’ll store this information in a HashMap. The parent class is named UntypedActor because we’ll treat all messages as Objects.

Akka Example

```
public void onReceive(
    Object message){
  String s = message.toString();
  String reply = "OK" ;
  if (s == "DUMP") {
   reply = seen.toString());
  } else {
   seen.put(s, new Date());
  }
  getContext().replySafe(reply);
}
```


We have to define the onReceive message that is declared abstract in UntypedActor. For simplicity, we'll just convert the message to a string. If it equals "DUMP", that's our signal to return a "dump" of the current state of the hash map, as a string. Otherwise, we add the message string to the hashmap as the key with the current time as the value. Then we send a reply to the caller, either the "dump" of the hash map or "OK".

Akka Example

```
public ActorExample {
 public static void main(… args) {
  ActorRef ar = actorOf(
     MemoryActor.class).start();
  for (String s: args) {
   Object r = ar.sendRequestReply(s);
   System.out.println(s+": "+r));
  }
  Object r=ar.sendRequestReply("DUMP");
   System.out.println("DUMP: "+r));
}}
```


Friday, April 12, 13

Finally, a main class to run it. It calls Akka's "actorOf" method to create an instance of MemoryActor and return a "reference" (a.k.a. handle) to it. This "handle-body" pattern is used so Akka can restart an actor if necessary, then update the reference to point to the new actor so the reference doesn't become stale!
We loop through the input arguments and send each one to MemoryActor, await the reply, then print it out.

Akka Example

```
$ java -cp … ActorExample \
I am a Master Thespian!
I: OK
am: OK
a: OK
Master: OK
Thespian!: OK
DUMP: {
am=Wed Jul 25 20:14:44 CDT 2011,
a=Wed Jul 25 20:14:44 CDT 2011,
Master=Wed Jul 25 20:14:44 CDT 2011,
Thespian!=Wed Jul 25 20:14:44 CDT 2011,
I=Wed Jul 25 20:14:44 CDT 2011}
```


Compile and run it with the arguments “I am a Master Thespian”. You get five lines with <string>: OK and a final line (which I’ve wrapped for better legibility, DUMP: <hash_map.toString>. Note that the hash map toString doesn’t preserve insertion order, which is the general case for hash maps.

For simplicity, we used *synchronous messages.* *Asynchronous* messages *scale better.*

Friday, April 12, 13

To simplify the example, I just used synchronous messages, but in a real app, you would use async messages, because they scale better.


Better Concurrency
Software
Transactional
Memory
104

ACID Transactions

- Atomicity
- Consistency
- Isolation
- Durability

You're familiar with ACID transactions, a central feature of relational databases.

ACID transactions ensure data integrity.

They have many benefits...

Manage *memory* with Transactions?

- Atomicity
- Consistency
- Isolation
- ~~Durability~~

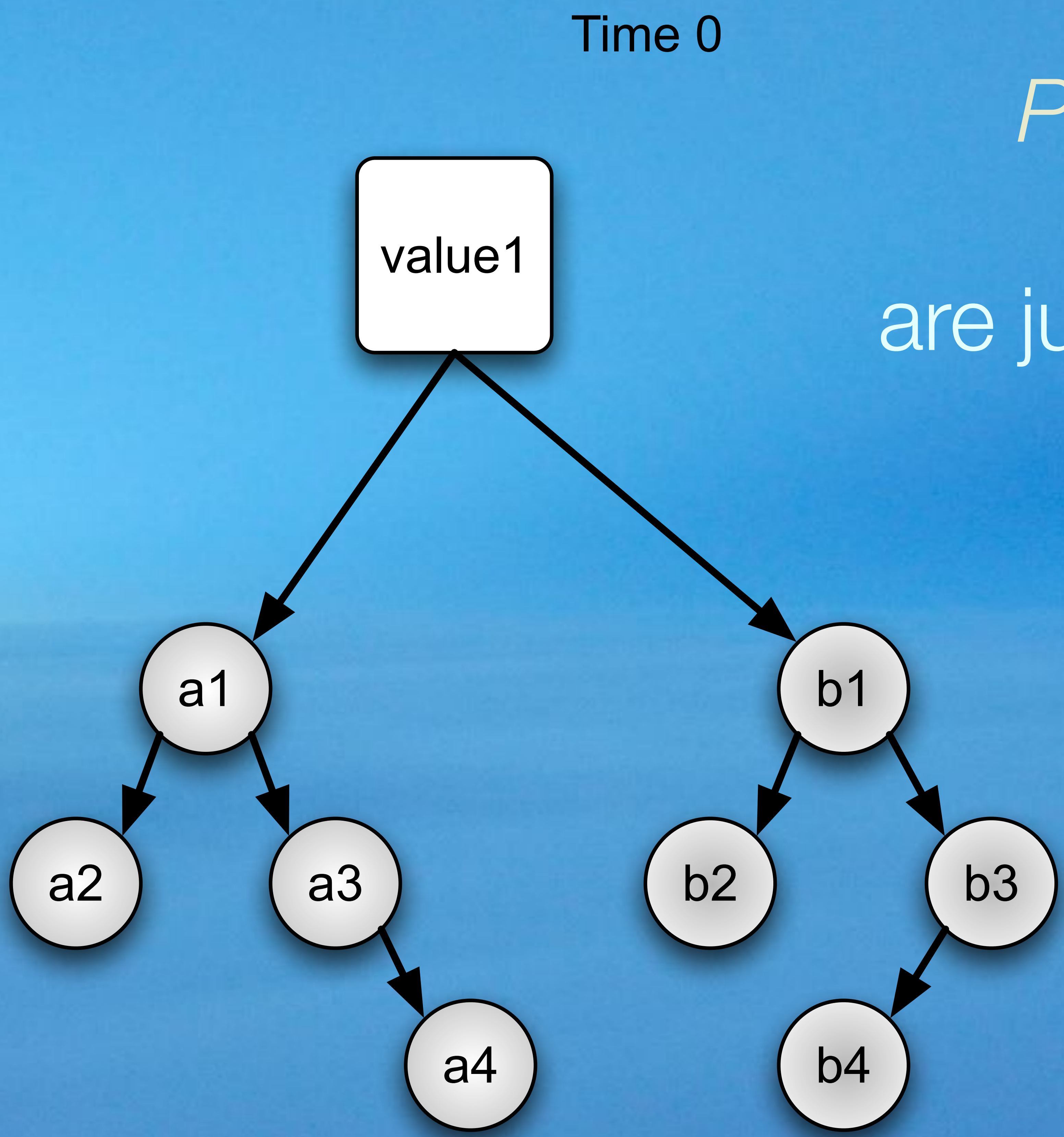
Friday, April 12, 13

Can we get the same semantics for updating values in memory?? Note that memory isn't durable.

Software Transactional Memory (STM)

- Atomicity
- Consistency
- Isolation
- ~~Durability~~

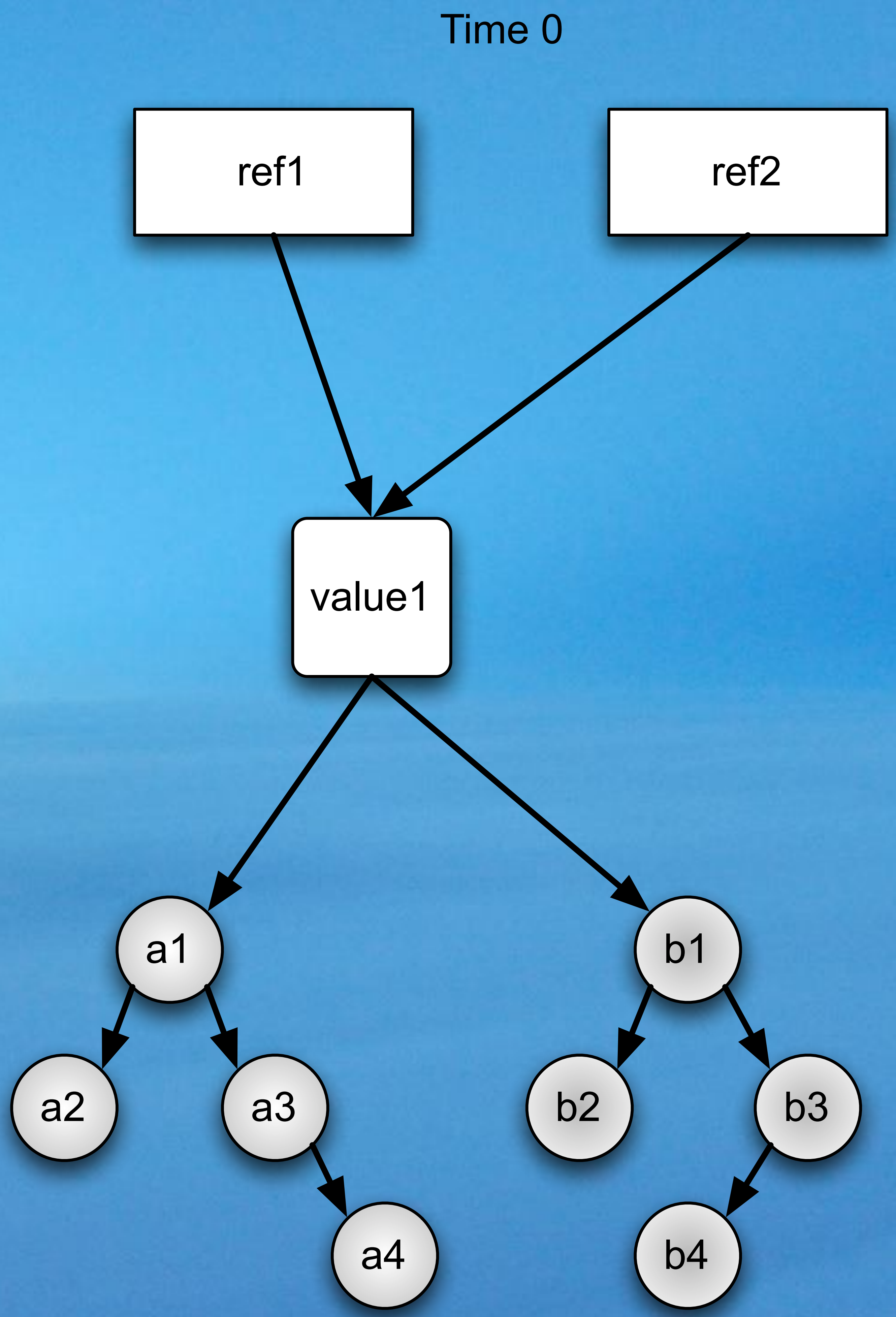
Software: it's managed in software (there were some experimental efforts to do this in hardware in the 90s).
Transactional semantics.
Memory: we're mutating values in memory.



Persistent Data Structures
are just what we need.

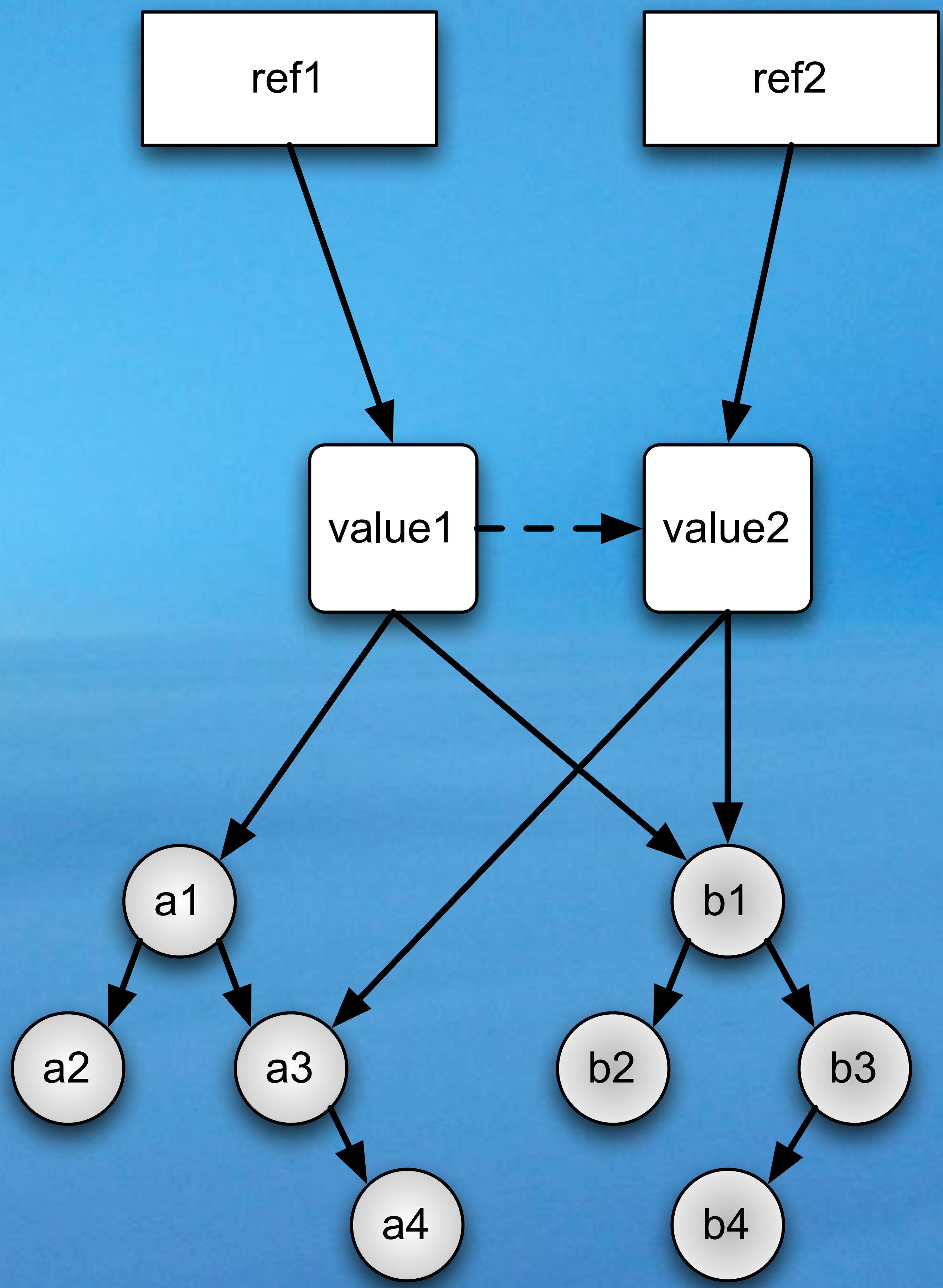
Friday, April 12, 13

Consider value1, which implements a Map, where the



At *time 0,* two references, *ref1* and *ref2* both refer to the same *value*.

Time 1

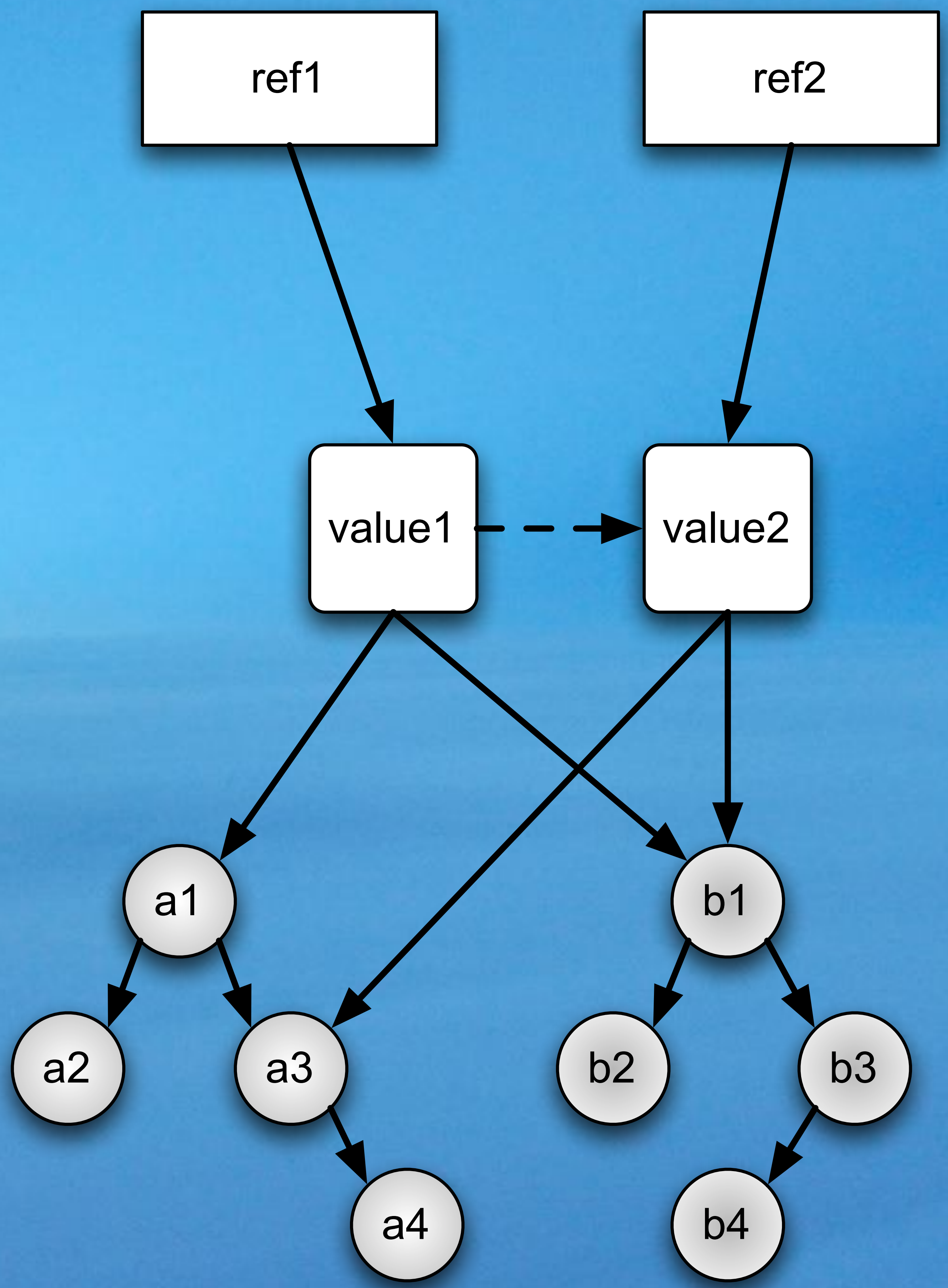


At *time 1*, *ref2* has been moved to the new *value*.

Friday, April 12, 13

A transaction is used to move the reference. Some APIs resemble the use of the synchronized keyword in Java. The transaction may include both the construction of value2 and the reassignment of ref2 to value2. However, since values are immutable, it's possible in this case to construct value2 first, then use a transaction to move ref2 to it.

Time 1



In *Clojure* simple assignment to *mutate* a value isn't allowed. STM is one of several mechanisms you must use.

Friday, April 12, 13

Clojure is making STM "famous".

The Haystack, Oregon

Better Objects


Immutable
Values
Better Objects
114

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

Friday, April 12, 13

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.

These techniques help you achieve immutability in any language. Persistent data structures let you make "copies" of big data structures efficiently.


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 value
 v = value.instance_of?(Money) ?
     value.value : value
 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!

Friday, April 12, 13

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.

Friday, April 12, 13

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

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
}
```

How might you *refactor* this code?

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?

Friday, April 12, 13

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.

Recall

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

Replace Conditional with Structure

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

We avoided conditionals in many list methods by using subclass polymorphism. That is we used good ol' OO-style refactoring.


Design
Patterns
Better Objects

Does FP make Design Patterns *obsolete*?

Friday, April 12, 13

Some people have claimed that FP makes design patterns obsolete. This confuses the idea of patterns with specific examples. There are some OO patterns that simply go away or are built into functional languages. Other OO patterns are still useful and FP has it's own collection of patterns, although the FP community has not traditionally used that terminology.

Some OO patterns go away:

Visitor

Good riddance!

Visitor is confusing, ugly, and invasive

Others are built into the FP languages:

Iterator, Composite, Command, ...

Friday, April 12, 13

Some other patterns are already in the language. Does that mean they *aren't* actually patterns?? Or, does it mean that we shouldn't think of patterns as something that *has* to be external to the language itself?

Others are new to FP:

Fold, Monoid, Monad, Applicative, Functor...

We saw *fold*. The others come from *Category Theory*...

Fold we saw. I'm just going to mention these Category Theory "patterns", but not define them. They're part of the intermediate material...

Visitor is replaced by *Pattern Matching* and less reliance on joining functions + state into objects.

Visitor is confusing, ugly, and invasive. It's designed to allow "visitors" to see object internals without simply exposing internals with getters. It's a way of adding (or simulating adding) new methods to existing classes for closes-type languages like Java.

The word "pattern" in "pattern matching" is not meant in the design pattern sense.

Pattern Matching is one of the most *pervasive* tools in functional programming.

Haskell/Erlang Like...

```
String toString(emptyList()){
 return "()";
}
String toString(list(head,tail)){
 return "("+head+","+tail+")";
}
...
List<X> list = new List<X>();
toString(list);
```


I’ve used Java syntax here, but this is the sort of code you see in Haskell and Erlang all the time, for example. A ListToString *module* would have multiple functions with the same name but different argument lists. The runtime picks the function by *matching* the argument to the first fit. AND it automatically extracts the head and tail for nonempty lists.
How does this work? Depending on the language, there would be a mechanism to *deconstruct* (or *destructure*) objects. Note that I’m showing our factory methods used in this way. So, there would need to be a “symmetry” defined in the language for this purpose. Scala uses a mechanism like this.

Haskell/Erlang Like…

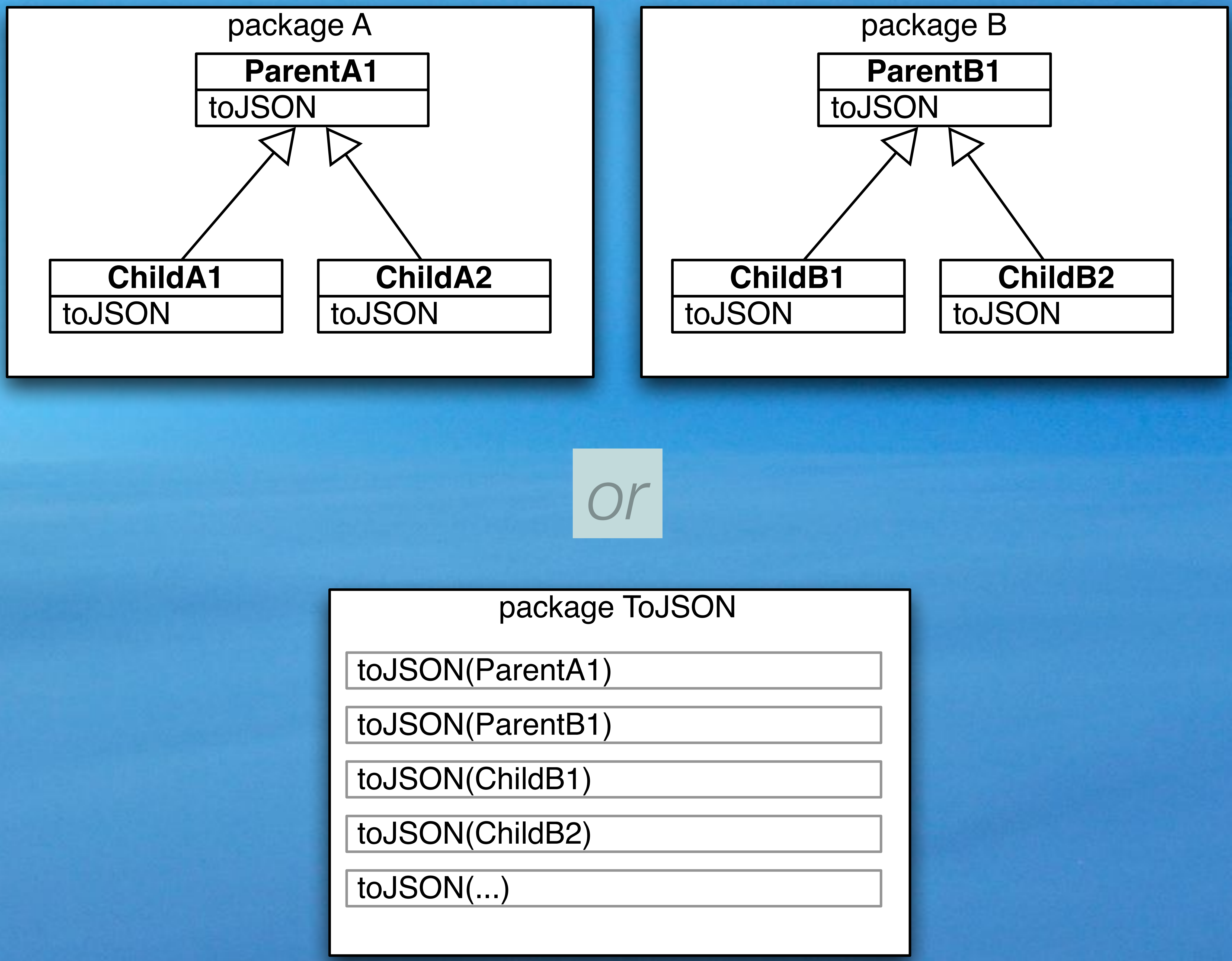
```
def to_s(List::EMPTY)
  "()"
end
def to_s(List(head,tail))
  "("+head+","+tail+")";
end
…
list = List.new(…)
to_s(list)
```


I’ve used “illegal” Ruby syntax here, but this is the sort of code you see in Haskell and Erlang all the time, for example. A ListToString *module* would have multiple functions with the same name but different argument lists. The runtime picks the function by *matching* the argument to the first fit. AND it automatically extracts the head and tail for nonempty lists.
How does this work? Depending on the language, there would be a mechanism to *deconstruct* (or *destructure*) objects. Note that I’m showing something that looks like a constructor call in the second example. So, there would need to be a “symmetry” defined in the language for this purpose. Scala uses a mechanism like this.

Wait!

Why am I defining to_s *outside* the classes??

Why IS to_s (or toString in other languages) in all objects? Yea, it's nice for debugging, but when is the format what you want? What if you want XML today and JSON tomorrow?



Friday, April 12, 13

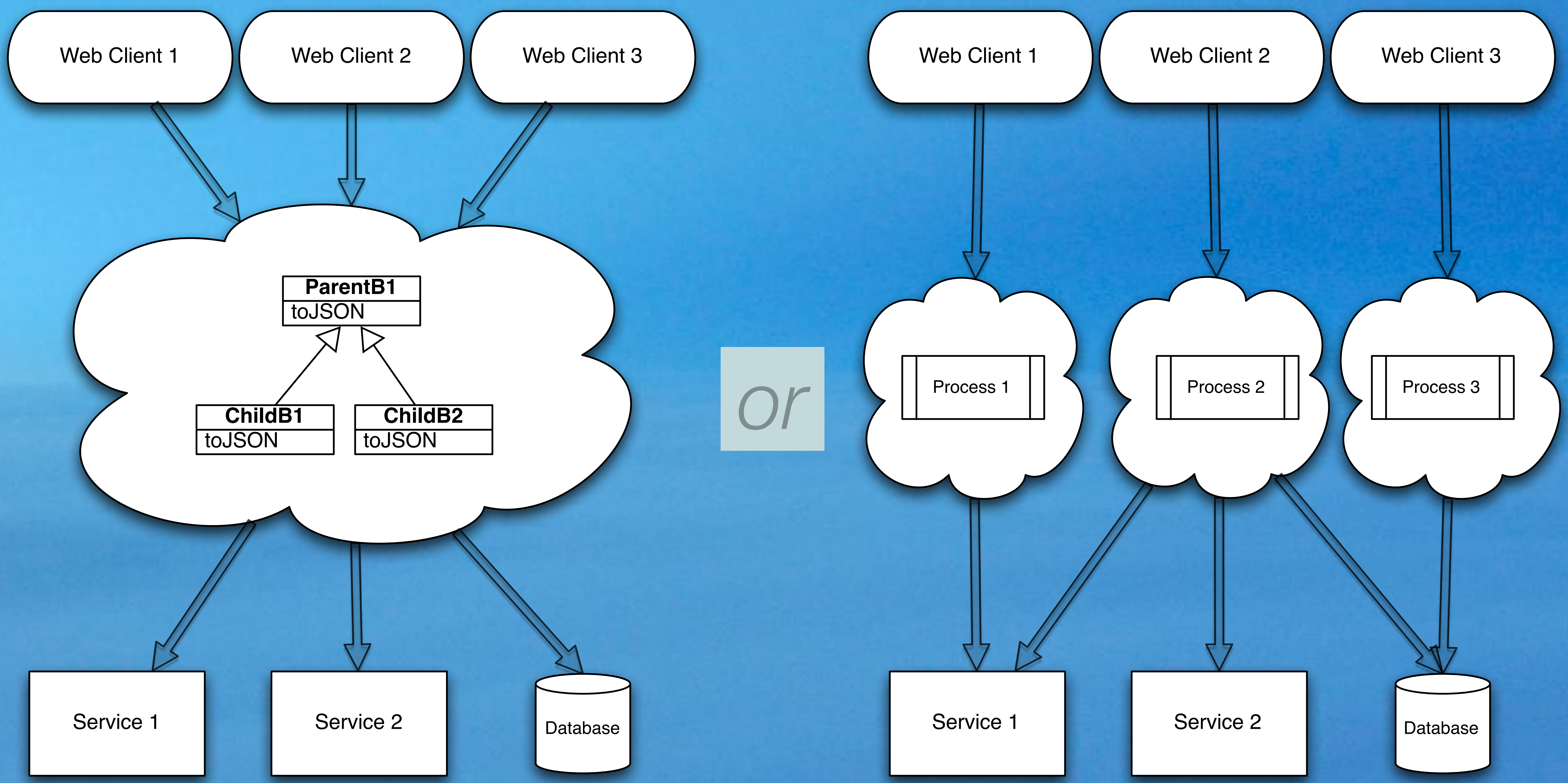
You really don't want to just bloat your classes with these things AND you want the *implementation* of "toJSON" for all types to be defined as modularly as possible. *I argue that putting stuff like this in class hierarchies all over you app scatters the logic and breaks modularity!

But doesn't "package ToJSON" break other rules? Like what if we add a new child to a hierarchy? We have to balance these competing design forces. For List, which is an Algebraic Data Type, this alternative works extremely well. For arbitrary hierarchies, it's more challenging.


OO
Middleware
Better Objects
139

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

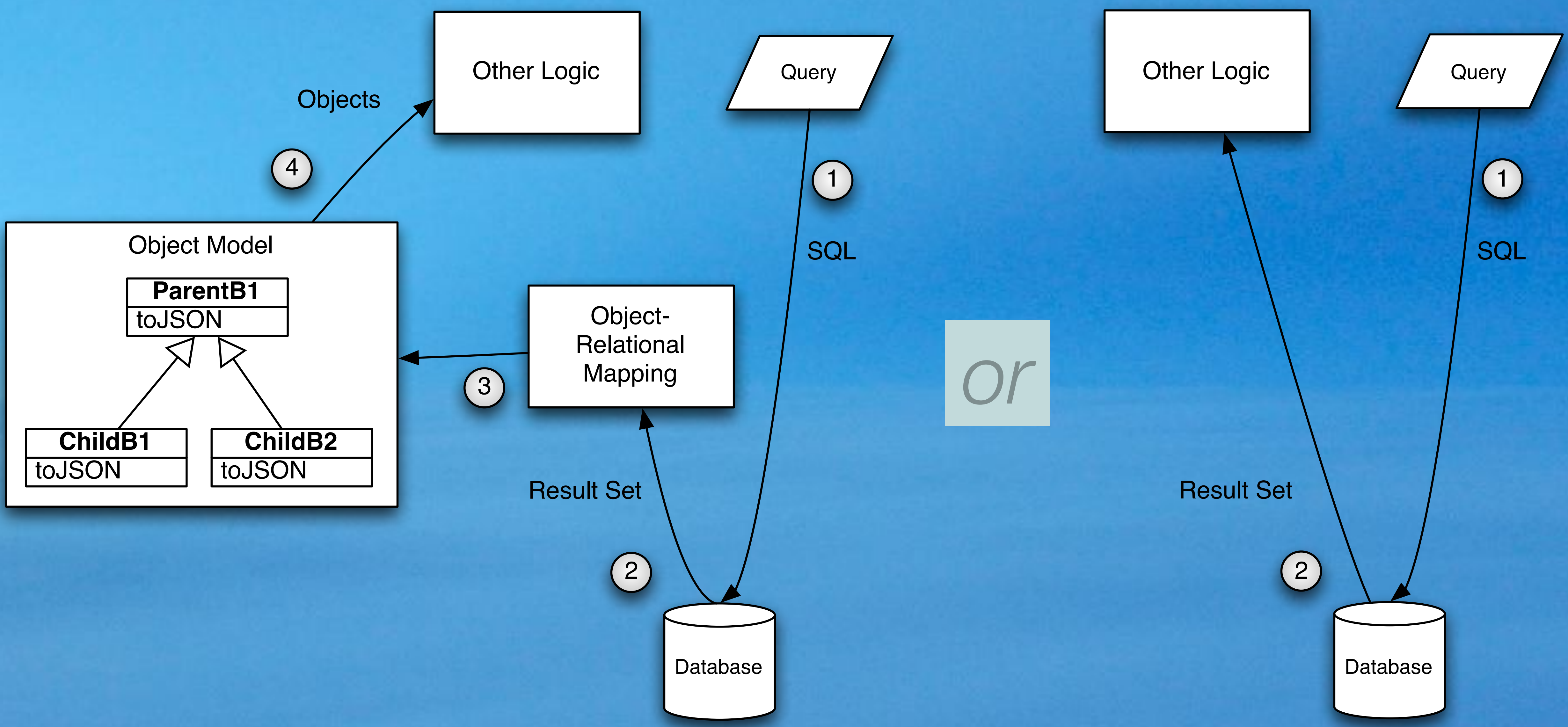
Which *Scales* Better?



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...

Friday, April 12, 13

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

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

Friday, April 12, 13

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

...

Friday, April 12, 13

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.

(Mud near Death Hollow in Utah.)

We need better agility.

Friday, April 12, 13

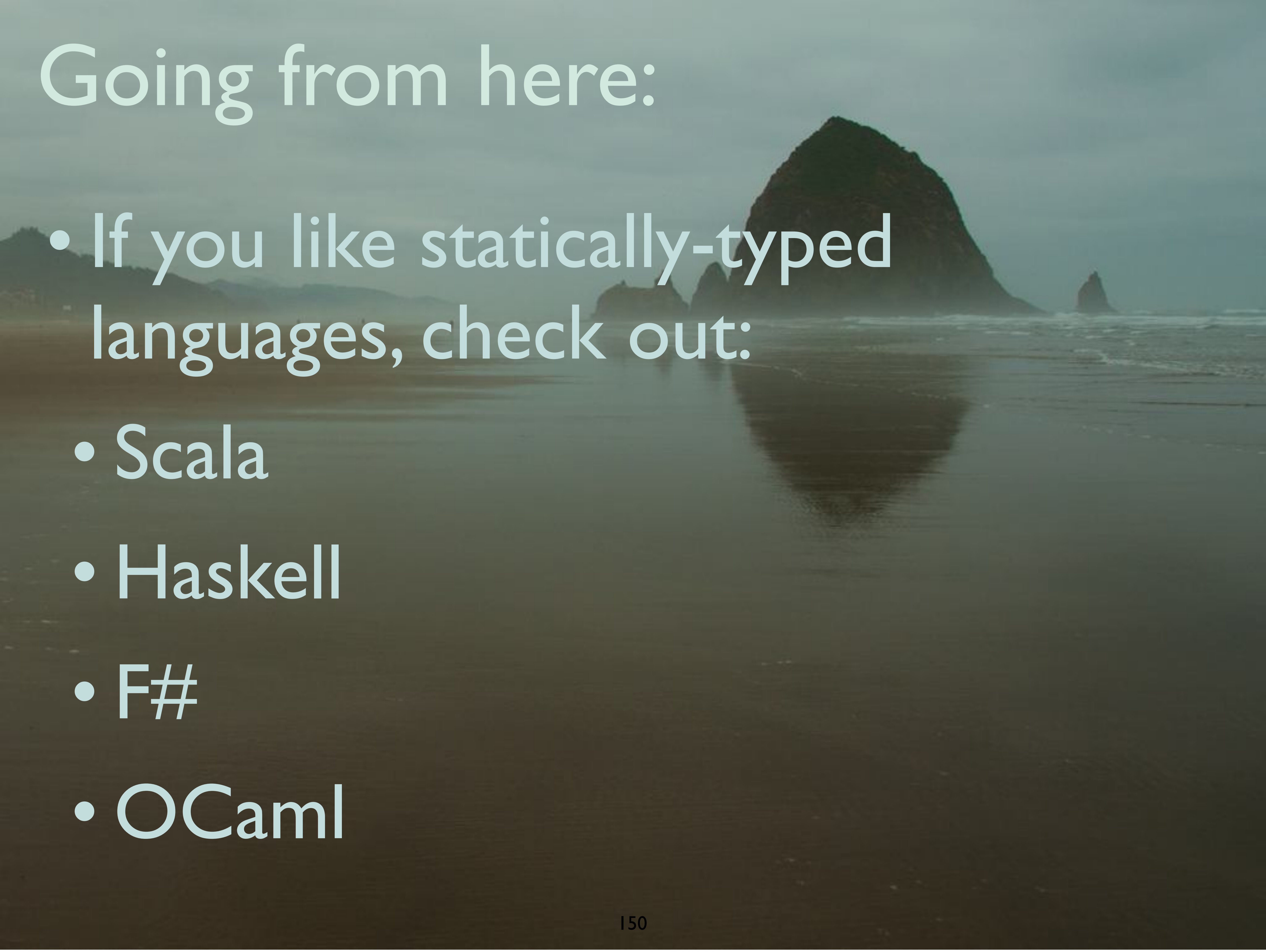
Schedules keep getting shorter. The Internet weeded out a lot of process waste, lot Big Documents Up Front, UML design, etc. From that emerged XP and other forms of Agile. But schedules and turnaround times continue to get shorter.

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

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. 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)



Friday, April 12, 13

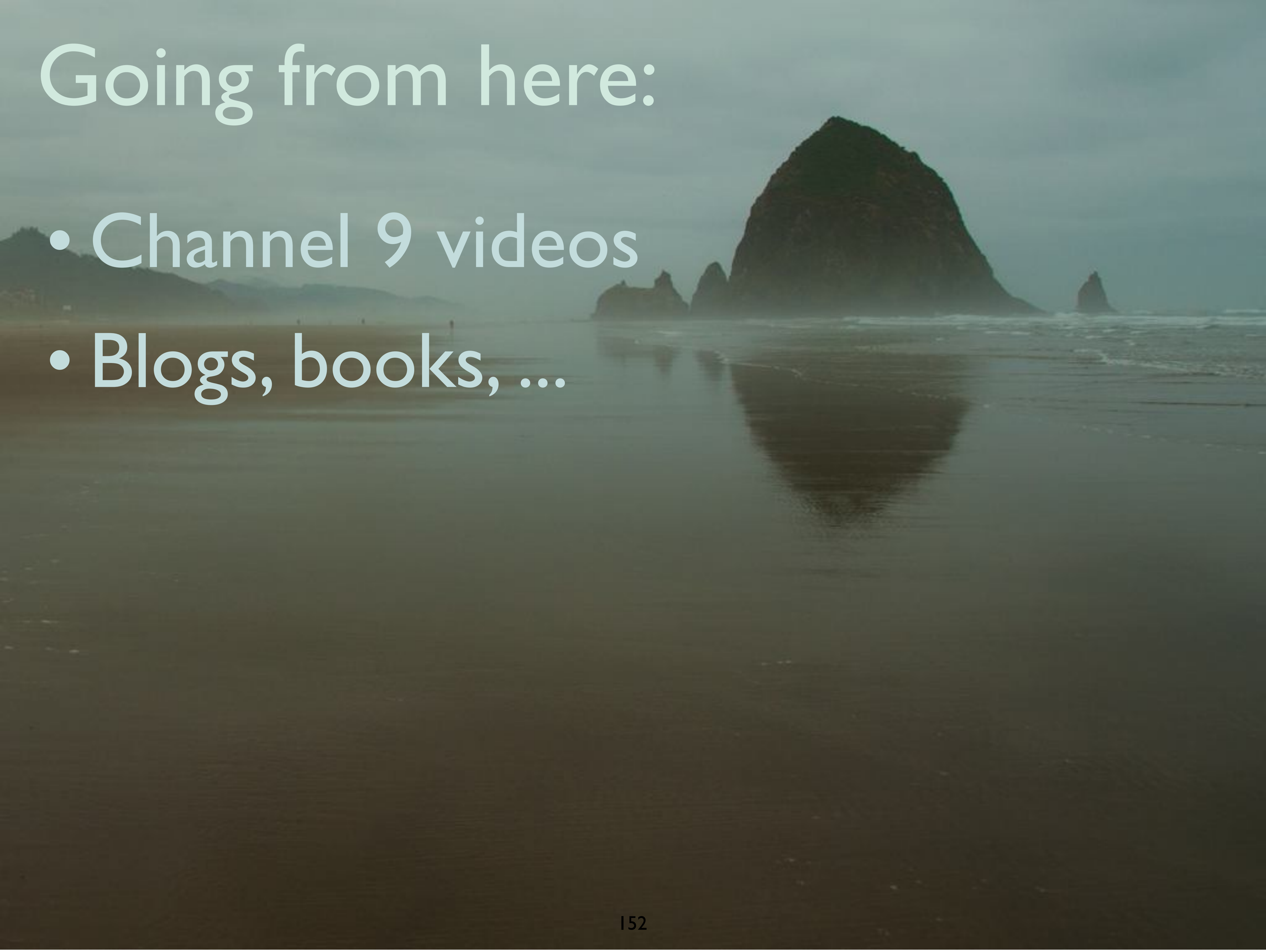
Learn a real functional language to see how these ideas work in a language that supports them natively, as well as concepts we didn't cover. Here is a list of the most popular statically-typed functional languages.

Going from here:

- If you like dynamically-typed languages, check out:
  - Clojure
  - Erlang
  - Other Lisp dialects

Friday, April 12, 13

Here is a list of the most popular dynamically-typed functional languages.



Friday, April 12, 13

There are excellent MSDN Channel 9 videos on functional programming.
Numerous blogs, books, etc...

Thank You!

- dean@deanwampler.com
- @deanwampler
- polyglotprogramming.com