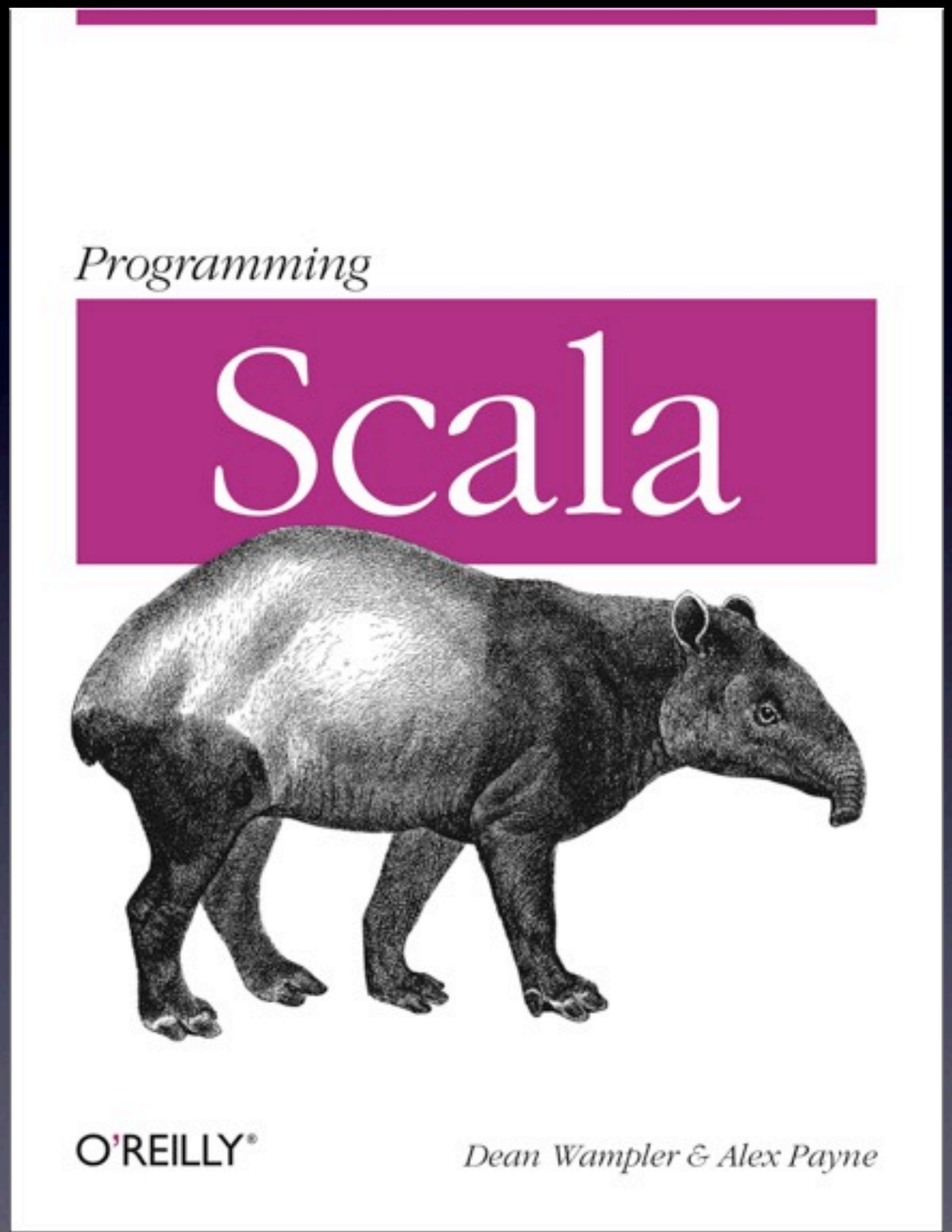
Better Ruby Through Functional Programming

Dean Wampler
dean@objectmentor.com
@deanwampler
polyglotprogramming.com/papers

I'm not just about Ruby...

Co-author,
Programming Scala

programmingscala.com

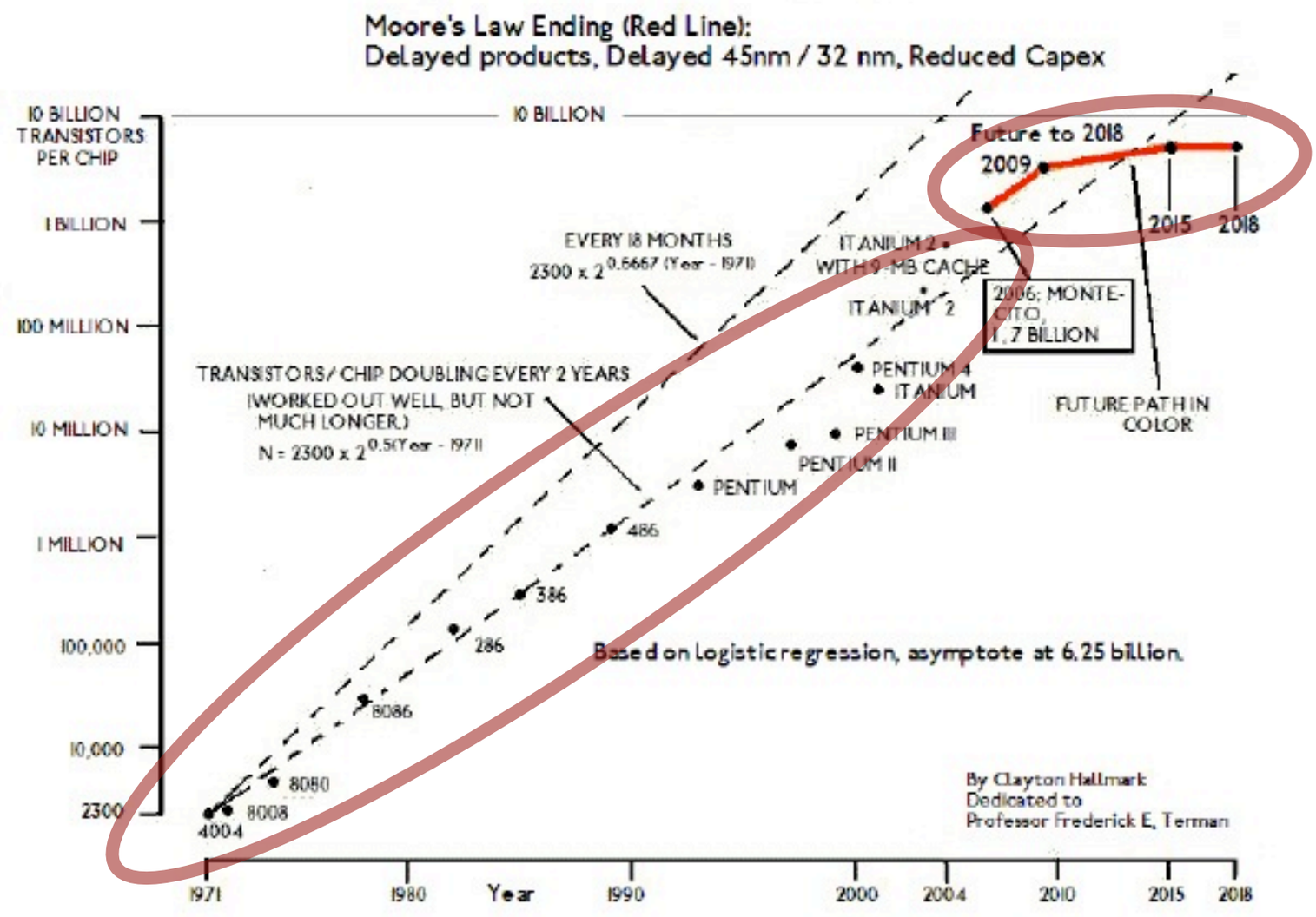


Saturday, May 30, 2009

Available now as a "Rough Cut" from oreilly.com.

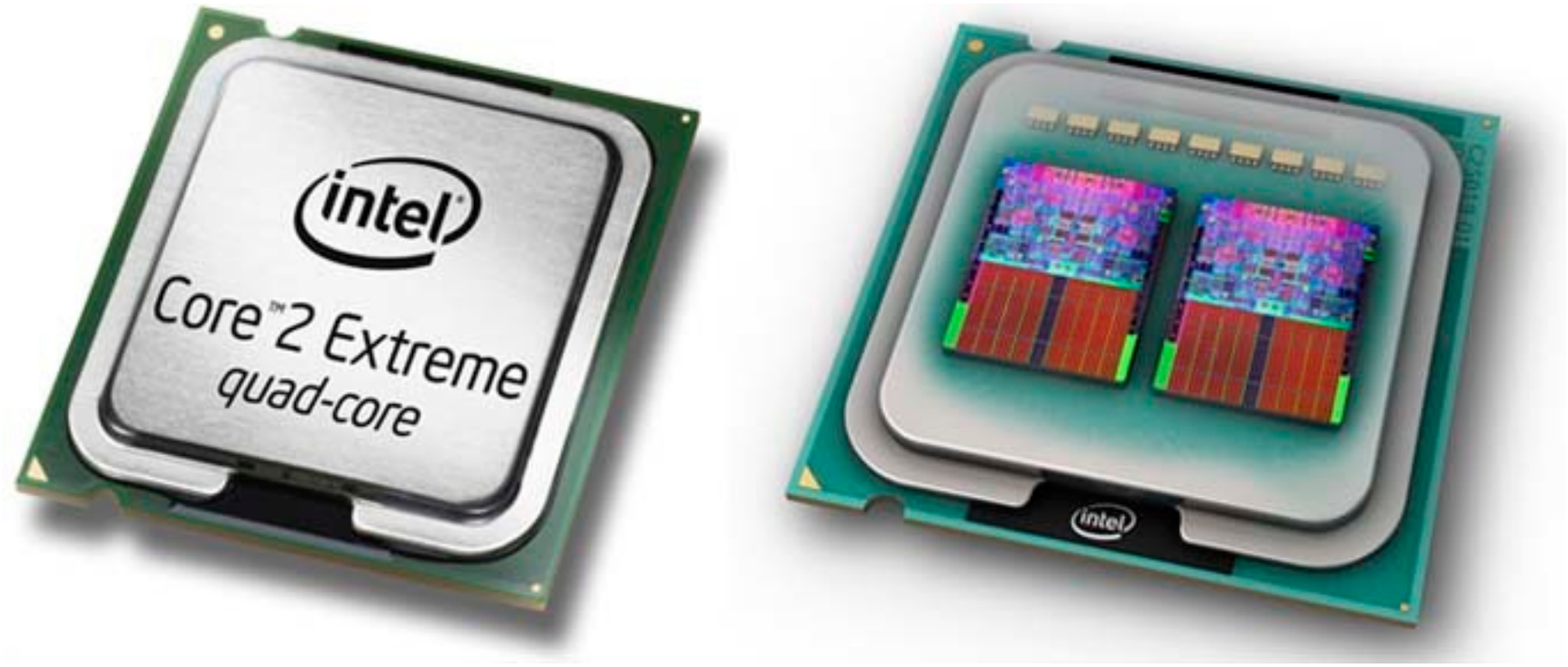
Why Functional Programming?

A story...



Saturday, May 30, 2009

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



so, we’re scaling horizontally.

How do we write *robust, concurrent* software??

Functional Programming!!

Saturday, May 30, 2009

That means writing robust, concurrent code, but this is hard in “imperative” languages. However, FP offers a solution.

What is *Functional Programming*?

Don't we already write "functions"?

$$y = \sin(x)$$

Based on Mathematics

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

$$y = \sin(x)$$

Setting *x* fixes *y*

∴ *variables* are *immutable*

Saturday, May 30, 2009

"Functional Programming" is based on the behavior of mathematical functions.
"Variables" are actually 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 state*

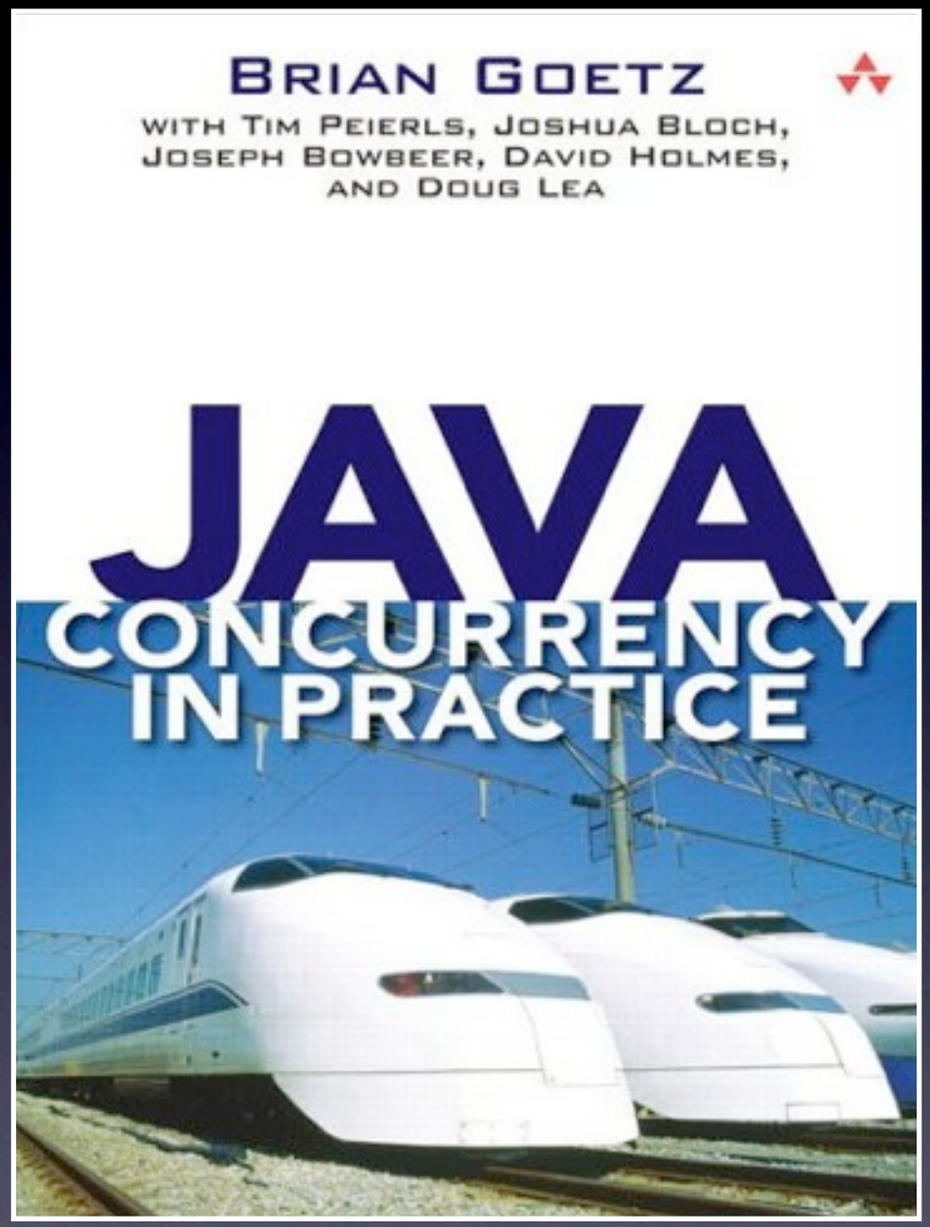
∴ *nothing* to *synchronize*

"No mutable state": There is changing state in a functional program. It's in the stack, though. 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 state...

Hic sunt dracones

(Here be dragons)

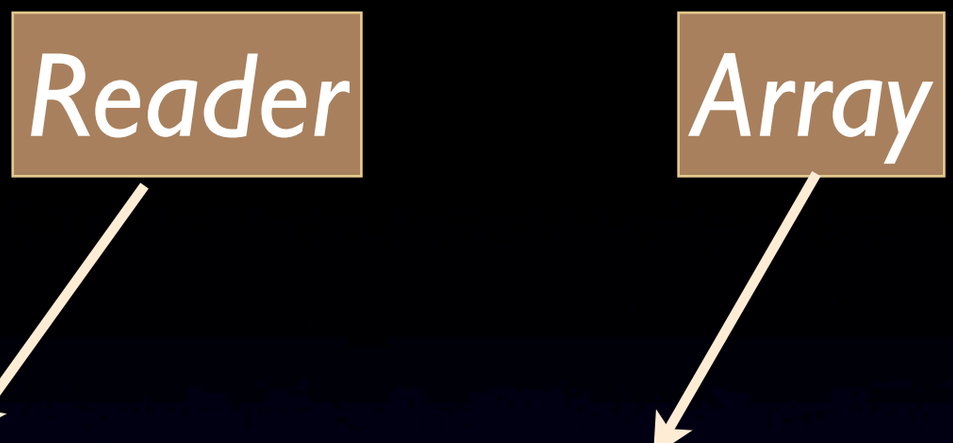


A *client* won't *mess up* an *immutable* object.

Another benefit...

Saturday, May 30, 2009

You can feel confident passing an immutable object to a “client” without fear that the client will modify the object in some undesirable way.




```
class Person
  attr_reader … :addresses
  def initialize …, addrs
    …
    @addresses = addrs
  end
end
```


Clients can’t change to a different array, but they can add and delete elements in the addresses!

$$y = \sin(x)$$

Functions don't

change state

$\therefore$ *side-effect* free

Saturday, May 30, 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 *reason*
about *behavior*.

Saturday, May 30, 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.

Saturday, May 30, 2009

Much less likely to have concurrency issues.

Side-effect free: Location *transparency.*

Saturday, May 30, 2009

Therefore, you can call it anywhere and the behavior is the same for the same inputs.

tan(x) = sin/cos(x)

Compose functions of other functions

∴ *first-class citizens*

Saturday, May 30, 2009

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

We'll see other facets of FP later...

Saturday, May 30, 2009

Ruby isn't a functional language, per se, so can we do functional programming in Ruby?


Functional
Programming
in Ruby

Immutable Types

Saturday, May 30, 2009

One way to be “pure” FP is to make your types immutable.

```
class Person
  attr_reader :first_name,
              :last_name,
              :addresses, …
  def initialize fn, ln, addrs
    @first_name = fn
    @last_name  = ln
    @addresses  =
      addrs.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".

```
dean = Person.new(
   "Dean", "Wampler", [“addr1”, …]
p dean.first_name # => "Dean"
p dean.addresses  # => ["addr1"…]

dean.first_name = "Bubba"
   # => … undefined method …
dean.addresses[0] = "new addr"
   # => … can't modify frozen …
```


First, we can support immutable types by using “attr_reader” only.

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

Caution!!

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 new Person

Saturday, May 30, 2009

Instead, return a new Person.

Tradeoff:

copy *overhead*

vs.

more *robustness.*

Maybe you don’t want all instances immutable, but just some of them...

Read-write attributes

```
class Person
  attr_accessor :first_name,
                :last_name, …
end

dean = Person.new "Dean","Wampler"
dean.first_name = "Bubba"
dean.freeze
dean.first_name = "Joe"
    # => … can't modify frozen …
```

freeze object

As an alternative, we can support immutable objects. That is, some instances of a type are immutable, while others aren't. Use the "freeze" method.

However:

What if an *attribute* is a *collection* or "*subobject*"?

Have to freeze it, too.

Module.freeze is "shallow". It doesn't freeze the contents of collections.

```
class Person
  attr_accessor …, :addresses
  def initialize …, addrs
    @addresses = addrs.clone
  end
  def freeze
    super
    @addresses.freeze
  end
end
```

Clone collections!

Override #freeze

Freeze fields of mutable types

Saturday, May 30, 2009

Override freeze to make it a “deep freeze”. The default freeze is shallow.
Should clone input addrs so if you freeze one object, you don’t “mysteriously” freeze collection used elsewhere.

```
dean = Person.new …,
   ["Wisteria Lane",
    “1 Vacation Dr.”]
dean.addresses[0] = "2 Home St."
dean.freeze
dean.addresses[0] = "3 House Dr."
    # => … can't modify frozen …
```


As an alternative, we can support immutable objects. That is, some instances of a type are immutable, while others aren’t. Use the “freeze” method.

However:

```
…
dean = Person.new "Dean","Wampler"
dean.freeze
dean = Person.new "Joe", "Plumber"
```

allowed

Saturday, May 30, 2009

However, you can reassign the variable to a new object. We're not freezing references, we're freezing what they point to.

Drawback:
Creation of *Lots* of Objects

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


Saturday, May 30, 2009

You'll create lots of copies of objects if they are immutable.

Immutable
objects
aren't *simple*

Saturday, May 30, 2009

It's not trivial making objects immutable. You have to be aware of the subtleties.

Side-Effect Free Functions in 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 do this in the language, but we can do it "manually". 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, "filter" is side-effect free, but what if the "array" is modified by another thread?

First-Class Functions in Ruby?

```
def predicate s
  s[0] == ?a
end

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


Saturday, May 30, 2009

Predicated: fancy name for a function returning true or false.
Example code...

```ruby
a = ["aa", "ab", "ba", "bc", "ac"]

a2 = filter(a) {|s| predicate s}
p a2          # => ["aa", "ab", "ac"]
```

```ruby
a3 = filter(a,&method(:predicate))
p a3          # => ["aa", "ab", "ac"]
```


Either have to wrap the function call in a block or use the &method(:foo) idiom.

```
a2 = filter(a) {|s| predicate s}

# The following won't work:
a2 = filter a, predicate
a2 = filter a, &predicate
```


You can't just pass the method name..., even the "proc-ized" form, because of Ruby's parse rules. Is it supposed to invoke the predicate with no arguments and no parentheses and pass the result or is supposed to pass the predicate function as a value?

First-Class
Functions in Ruby?

Procs but not *methods*

Saturday, May 30, 2009

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

Saturday, May 30, 2009

Of course, the concept is actually named after Haskell Curry, the great Mathematician and Logician who pioneered combinatory logic, among other things.

Modulo Operation

mod(m)(n) = n % m

mod2 = mod(2)

mod2(5) = 1

An example of a curried function.

Currying
made possible by
higher-order functions.

$$\mathrm{mod}(m)(n) = f(n, g(m))$$

$$= n \,\%\, (m)$$

$$= n \,\%\, m$$

Saturday, May 30, 2009

Nested functions.

```ruby
mod = lambda {|m, n| n % m}
mod.(2, 5)  # ==> 1

modcurry = mod.curry
mod2 = modcurry.(2)
mod2.class # ==> Proc
mod2.(5)   # ==> 1
```

```ruby
mod25 = modcurry.(2, 5) # 1
mod25.class # ==> Fixnum
```

Ruby 1.9

Saturday, May 30, 2009

Example of a curried function.


Recursion

Recall:

not allowed in "pure" FP

```
for i in [1, 2, 3, 4, 5]
  print "#{i}, "
end
```


In pure FP, traditional for loops are not allowed, because they must change the value of the loop counter! So recursion is more important in FP than OOP/imperative.

Prefer:

```
[1, 2, 3, 4, 5].each do |i|
  print "#{i}, "
end
```

"i" is "new" every time.

Saturday, May 30, 2009

In pure FP, traditional for loops are not allowed, because they must change the value of the loop counter! So recursion is more important in FP than OOP/imperative.

Use Recursion:

```
def print_array array, i = 0
  return if i == array.length
  print "#{array[i]}, "
  print_array array, i + 1
end
```

No mutable data. No side effects.

But, is this better than a loop?

Saturday, May 30, 2009

So, we use recursion instead. Note that no variable is modified here. (However, technically, the print statement is not side-effect free, since it “modifies the environment”.) However, the clarity isn’t necessarily better than the imperative for loop.

Recursion:

Can *blow* the *stack*.

Ruby 1.9.1 has tail-call optimization

Two drawbacks. Ruby 1.9.1 does tail-call optimization.

This is a tail call

```
def print_array array, i = 0
  return if i == array.length
  print "#{array[i]}, "
  print_array array, i + 1
end
```

call to print_array is the last thing done.

Saturday, May 30, 2009

Not all recursive calls are “tail calls”. It’s good to learn to recognize them.

Recursion:
Is a *loop* more *clear*?

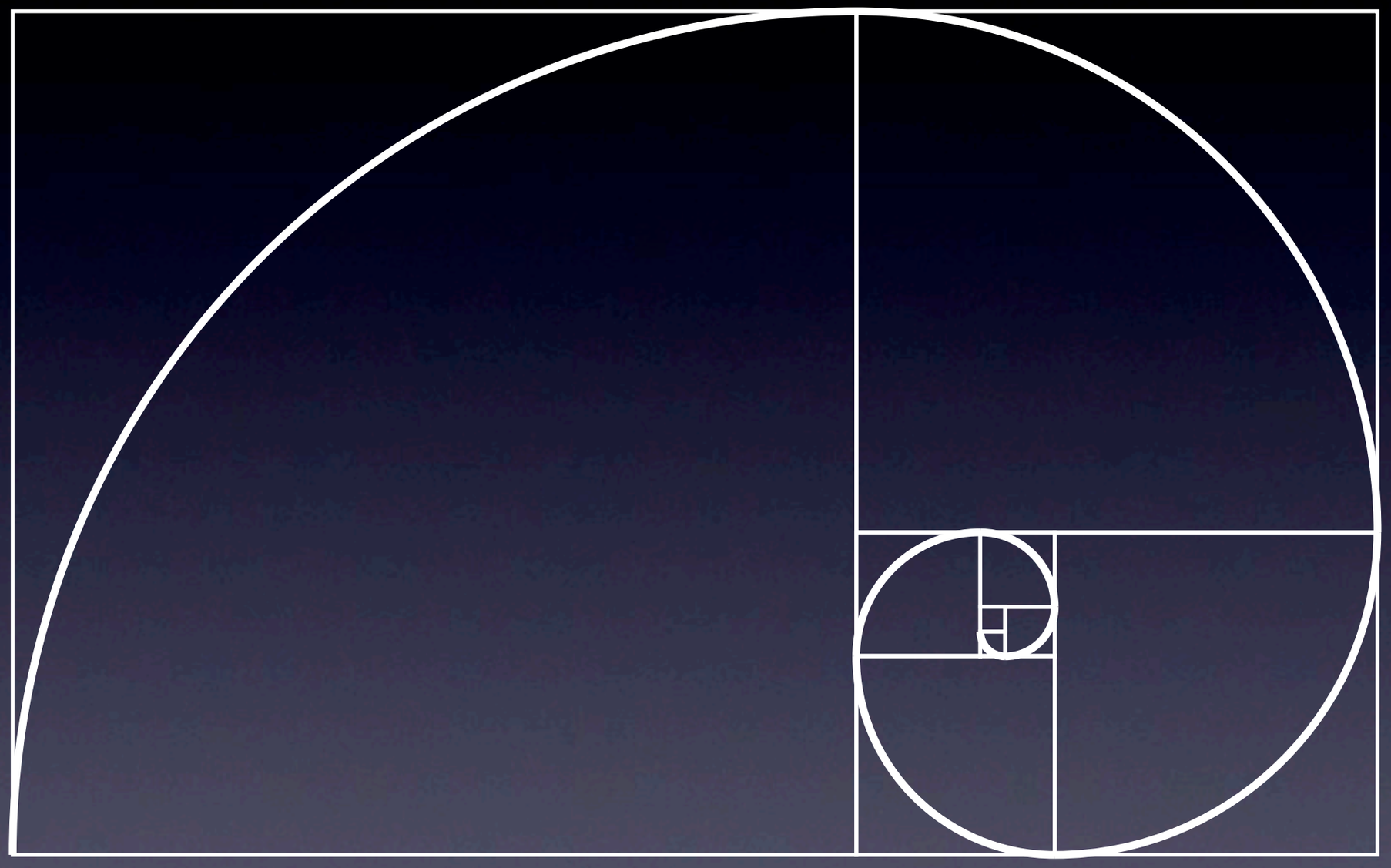
Saturday, May 30, 2009

Some problems are more naturally expressed as recursions.

Another example of *recursion*.

Some problems are more naturally expressed as recursions.

Fibonacci Spiral



From Wikipedia

Saturday, May 30, 2009

Let’s look at a nicer example of recursion.
This spiral appears a lot in nature, e.g., the shell of a nautilus.

Fibonacci Sequence

$$F(n) = \begin{cases} 0, \text{ if } n = 0 \\ 1, \text{ if } n = 1 \\ F(n-1) + F(n-2), \text{ if } n > 1 \end{cases}$$

Saturday, May 30, 2009

A better example of a recursive mathematical function.

```
def fib n
  case n
  when 0..1 then n
  else fib(n-1) + fib(n-2)
  end
end
```

case matching

Not a tail call

Function looks like math definition!

Saturday, May 30, 2009

An elegant definition. Note that we are using a case statement. We'll return to "pattern matching" later.

OO code is more *imperative*.

```
…
p = Person.new(…)
p.setJobTitle(…)
p.pay(p.getSalary()/26 +
  salesTeam.getQuarterlyBonus())
…
```


Saturday, May 30, 2009

Imperative code: I tell you how to do the work.
Declarative code: I tell you what I want and you figure it out.

Functional programs are very *declarative*.

```
def fib n
  case n
  when 0..1 then n
  else fib(n-1) + fib(n-2)
  end
end
```


Saturday, May 30, 2009

Imperative code: I tell you how to do the work.
Declarative code: I tell you what I want and you figure it out.

Side-effect free

```
def fib n
  case n
  when 0..1 then n
  else fib(n-1) + fib(n-2)
  end
end
```

But what if values are cached internally?

Saturday, May 30, 2009

Be careful to preserve “referential transparency”. This function has side-effect free semantics. It should always return the same value for a given “n”, no matter when or where it’s called. However, for optimization, previous values might be cached, which is a side effect. Implementer has to preserve semantics...

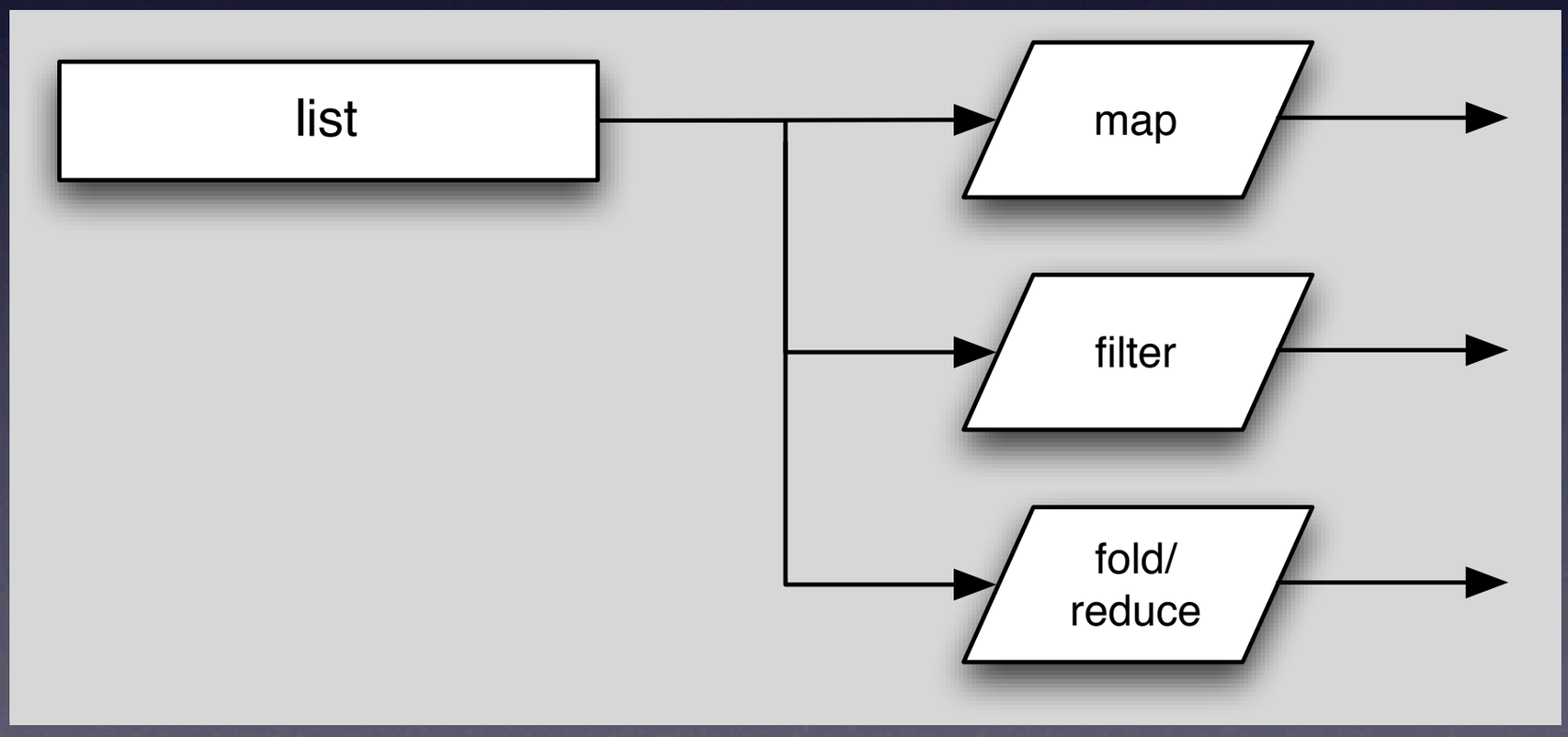
While *side-effect free*
on the outside,
it might *cache*
values for *performance*

Saturday, May 30, 2009

The implementer has to be careful to preserve side-effect free behavior, e.g., ensure thread safety in the caching code.

Functional Data Structures

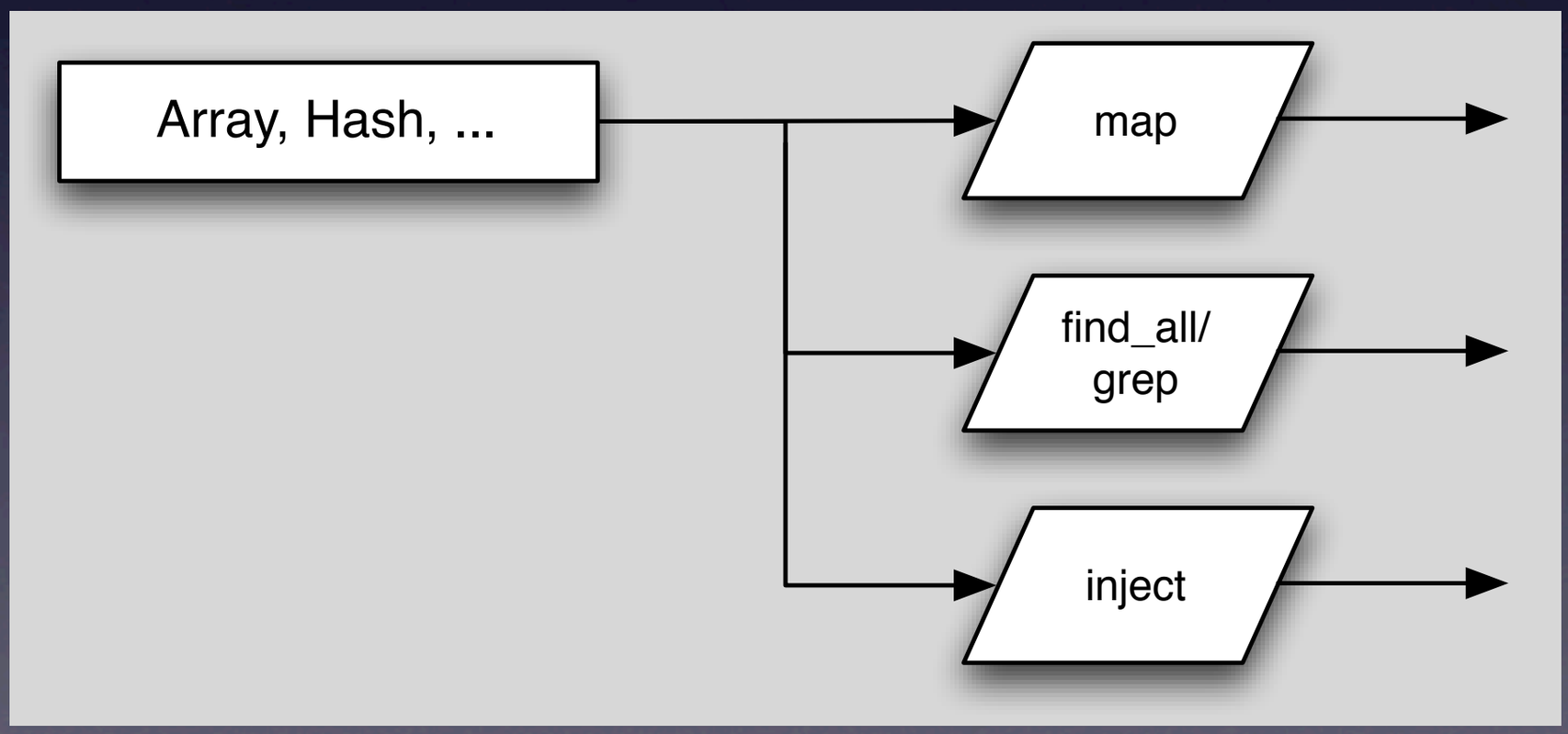
Classic Operations on *Functional Data Types*



Saturday, May 30, 2009

List being the most commonly-used data type.

In Ruby



Ruby equivalents.

Note:

Object trees reduce to *primitives* and *collections*.

Any arbitrarily-complex object graph reduces recursively to primitives (numbers, strings, ...) and collections of things.

∴ You can apply *external* functions to *primitives* and *collections* to implement *behaviors*.

Saturday, May 30, 2009

Not all behaviors belong in the objects. We can “compose” the behaviors we need by iterating through collections in our objects and applying blocks to them.

I.e., which *behaviors* should be *methods vs. external*?

Saturday, May 30, 2009

Not all behaviors belong in the objects. We can “compose” the behaviors we need by iterating through collections in our objects and applying blocks to them.

Recall:

```
dean = Person.new …,
         "1 Memory Lane",
         "1 Hope Drive",
         "1 Infinite Loop"
dean.addresses.each {|a| puts a}
dean.addresses.sort! {|a,b| a<=>b}
      # fail!
```


We can iterate through the addresses to use them, but we can't modify them.

```
def validate address
   raise "…" unless
     in_USPS_database?(address)
end

…

dean.addresses.each do |a|
   validate a
end
```

One example…

Saturday, May 30, 2009

Apply our functional operations to our objects. Example #1: validate the addresses. This is not the sort of code you necessarily want in your “pristine” domain model object, but it’s great as an external “service”.

```
def show_on_map address
  on_google_maps(address)
end
…
dean.addresses.each do |a|
  show_on_map a
end
```

Another example...

Saturday, May 30, 2009

Apply our functional operations to our objects. Example #2: Show the person's addresses on Google Maps.

Macro level: *OOP*

Micro level: *FP*

??

How it seems to go, in practice...

Saturday, May 30, 2009

Mixed paradigm apps organize themselves into objects, but the objects tend to be immutable with side-effect free (functional-style) methods.

Declarative

Programs

Recall:

```
def fib n
  case n
  when 0..1 then n
  else fib(n-1) + fib(n-2)
  end
end
```

Functional code is declarative.

Saturday, May 30, 2009

FP code tends to be declarative, since it isn't imperative. This makes it more concise, which is good for productivity, bug reduction, maintenance, etc. Also, it lets the "internals" decide the best way to implement what you want.

DSL's are *Declarative*, too

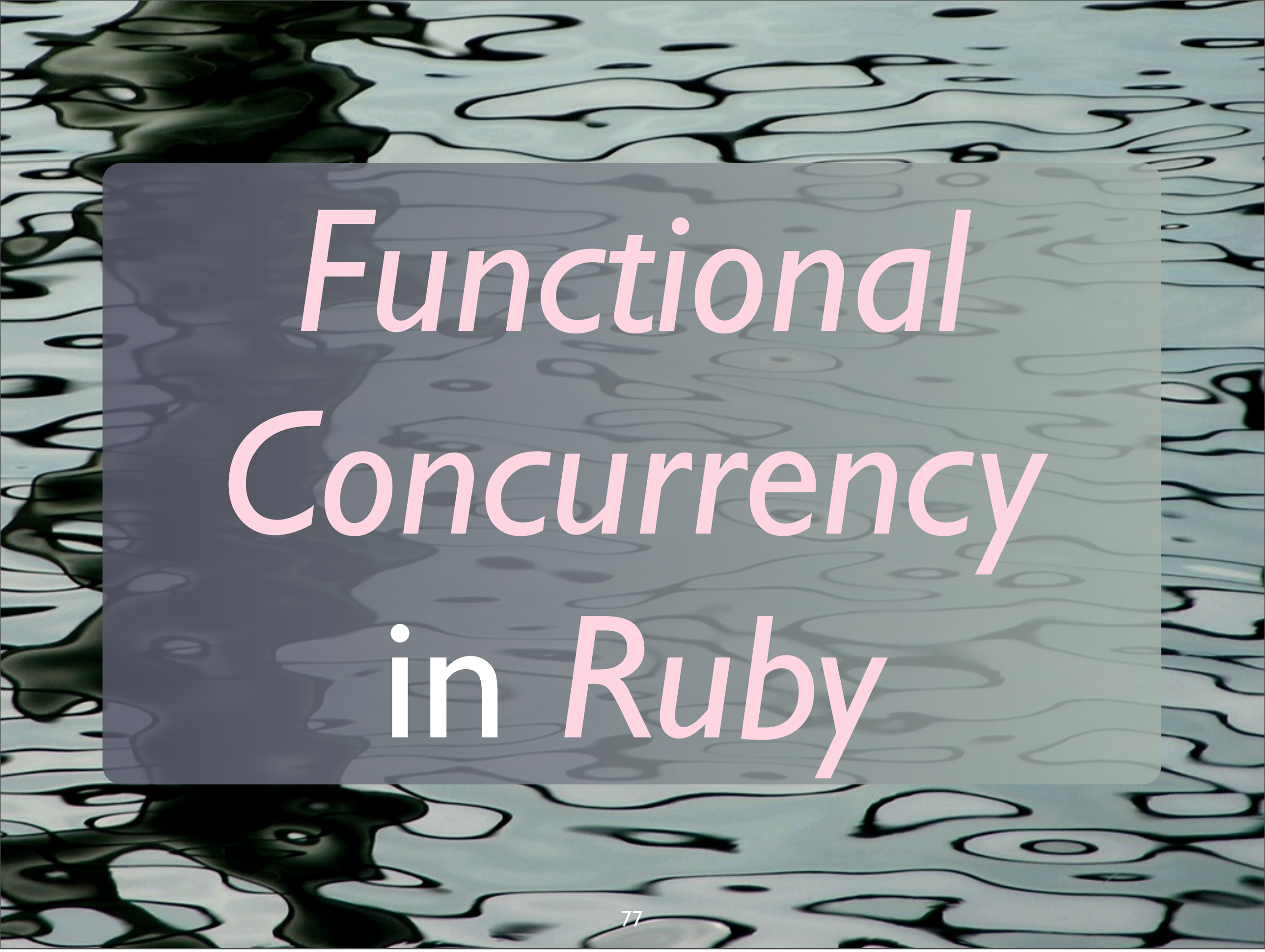
```
class Person < ActiveRecord::Base
  belongs_to :family
  has_many   :addresses
end
```

Active Record

Saturday, May 30, 2009

Ruby DSL's also have this quality. You say what you want and let ActiveRecord decide how to do it.

We all know that
DSL's are *good*...
Functional programming
has many of
the same *qualities*.


Functional
Concurrency
in Ruby

Actor Model of *Concurrency*

Autonomous *agents* exchange *messages*.

Finally, let's see an example of writing concurrent applications using actors. Actors don't share any state. They communicate only through messages. However, the actor might have its own self-contained state.

```
class Shape; end
```

Don't really need this...

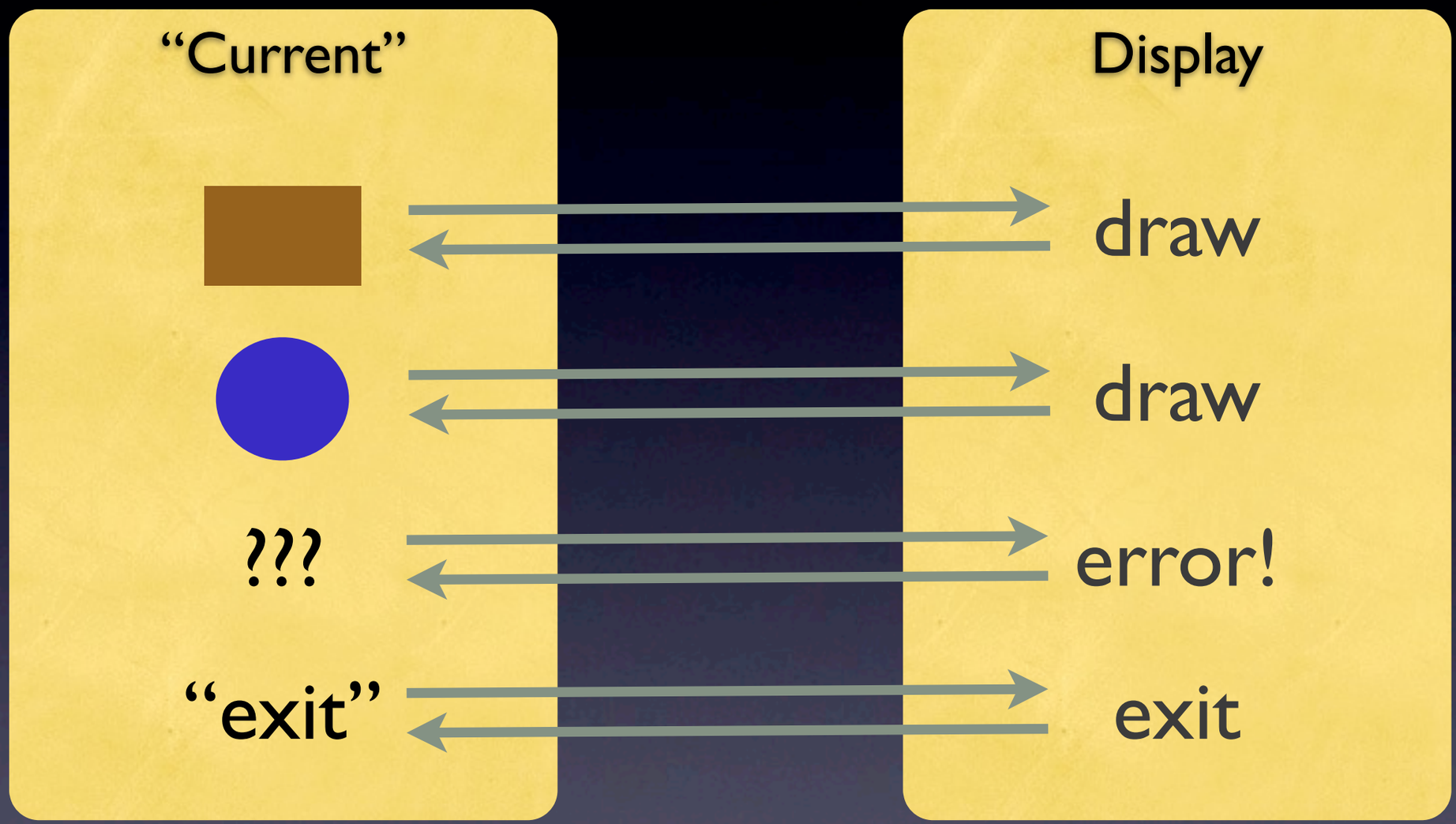
```
class Rectangle < Shape
  def draw
    "drawing Rectangle"
  end
end
```

Geometric Shapes

```
class Circle < Shape
  def draw
    "drawing Circle"
  end
end
```


Start with a hierarchy of geometric shapes.

2 Actors:



Omnibus Concurrency

```
require "rubygems"
require "concurrent/actors"
require "case"
require "shapes"
```

```
include Concurrent::Actors
...
```


Saturday, May 30, 2009

This example uses MenTaLguY's Omnibus Concurrency gem. See also Revactor (Ruby 1.9) and Dramatis. The Greeting and RequestGreeting objects will be used as the messages sent between actors.

```
…
Message =
   Struct.new :reply_to,:body

def make_msg body
    current = Actor.current
    Message.new current, body
end
…
```

"This" actor

This example uses MenTaLguY's Omnibus Concurrency gem. See also Revactor (Ruby 1.9) and Dramatis. The Greeting and RequestGreeting objects will be used as the messages sent between actors.

```
…
display = Actor.spawn do
 loop do
  Actor.receive do |msg|
   msg.when Message[Object,Shape] do |m|
    s = m.body.draw
    m.reply_to << "drew: #{s}"
   end
   msg.when Message[Object,:exit] do|m|
    m.reply_to << "exiting"
   end
   msg.when Message[Object,Object] do|m|
    m.reply_to << "Error! #{m.body}"
   end; end; end; end
…
```


Here is the main Actor code. An actor is “spawned” (e.g., in a separate Thread). It loops forever waiting to receive messages. It matches each message by type, using MenTaLguY’s Case gem, which enhances #===. Note the combination of pattern matching and polymorphism (Shape.draw).

```
…
display = Actor.spawn do
  loop do
    Actor.receive do |msg|
      msg.when Message[Object,Shape] do |m|
        s = m.body.draw
        m.reply_to << "drew: #{s}"
      end
      msg.when Message[Object,:exit] do|m|
        m.reply_to << "exiting"
      end
      msg.when Message[Object,Object] do|m|
        m.reply_to << "Error! #{m.body}"
      end; end; end; end
…
```

Case gem: #===

messages

Here is the main Actor code. An actor is “spawned” (e.g., in a separate Thread). It loops forever waiting to receive messages. It matches each message by type, using MenTaLguY’s Case gem, which enhances #===. Note the combination of pattern matching and polymorphism (Shape.draw).

```
…
display = Actor.spawn do
 loop do
  Actor.receive do |msg|
   msg.when Message[Object,Shape] do |m|
    s = m.body.draw
    m.reply_to << "drew: #{s}"
   end
   msg.when Message[Object,:exit] do|m|
    m.reply_to << "exiting"
   end
   msg.when Message[Object,Object] do|m|
    m.reply_to << "Error! #{m.body}"
   end; end; end; end
…
```


Here is the main Actor code. An actor is “spawned” (e.g., in a separate Thread). It loops forever waiting to receive messages. It matches each message by type, using MenTaLguY’s Case gem, which enhances #===. Note the combination of pattern matching and polymorphism (Shape.draw).

Send messages

```
…
display << make_msg Rectangle.new
display << make_msg Circle.new
display << make_msg "Hi, Display!"
display << make_msg :exit
…
```


Saturday, May 30, 2009

Here is the code in action.

```
…
loop do
 Actor.receive do |reply|
  reply.when "exiting" do |s|
    p "exiting..."
  end
  reply.when String do |s|
    p "reply: #{s}"
  end
end
```

Receive replies

Saturday, May 30, 2009

Here is the code in action.

```
"reply: drew: Rectangle"
"reply: drew: Circle"
"reply: Error! Hello Display!"
"exiting..."
```


What gets printed.

Pattern matching
is used in *FP*
like *polymorphism*
is used in *OOP*.

The actor example shows functional-style pattern matching, which is used heavily in FP in an analogous way that polymorphism is used in OOP.

```
…
display = Actor.spawn do
 loop do
  Actor.receive do |msg|
    msg.when Message[Object,Shape] do |m|
      s = m.body.draw
    ...
end
...
```

FP pattern matching…

… with OO polymorphism

A powerful combination!

Saturday, May 30, 2009

Note the combination of pattern matching and polymorphism (Shape.draw) is very powerful.

Lessons Learned

Prefer

immutability.

Saturday, May 30, 2009

Immutable variables are not only better for concurrency, but they are better in general. You don’t mind passing an immutable object to someone else, because they can’t screw it up!!

OOP and FP *complement each other*

Immutable variables are not only better for concurrency, but they are better in general. You don't mind passing an immutable object to someone else, because they can't screw it up!!

Macro level: *OOP*

Micro level: *FP*

??

How it seems to go, in practice...

Saturday, May 30, 2009

Mixed paradigm apps organize themselves into objects, but the objects tend to be immutable with side-effect free (functional-style) methods.

Prefer

side-effect free

functions.

Saturday, May 30, 2009

They are much easier to reason about, use (concurrently or not), etc. Note that this may be the externally-visible behavior. Internally, it may NOT be side-effect free, e.g., if it caches previously computed values.

What *behaviors* go *inside* a type vs. *outside*?

Saturday, May 30, 2009

Think about what behavior is “integral” to an object vs. “optional/incidental”.

Watch overhead for *copies* and *recursion*.

Saturday, May 30, 2009

Excess copying increases garbage collection and memory overhead. Recursion can blow the stack!!

Mix

pattern matching

and

polymorphism.

Saturday, May 30, 2009

Homework! Investigate how to use pattern matching AND polymorphism. Which is better in which contexts?

Learn

Erlang, Haskell, Clojure, Scala, F#, …

Finally, even if you never use any other language, I recommend you learn a functional language (Erlang, Haskell, OCaml, Scala), because the ideas you learn will inform your Ruby (and Java, Groovy, C#, …) code!

Thanks!

dean@objectmentor.com
@deanwampler

polyglotprogramming.com/papers
programmingscala.com