meta-programming /j?ruby/ for fun & profit

NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
blog: memeagora.blogspot.com
twitter: neal4d

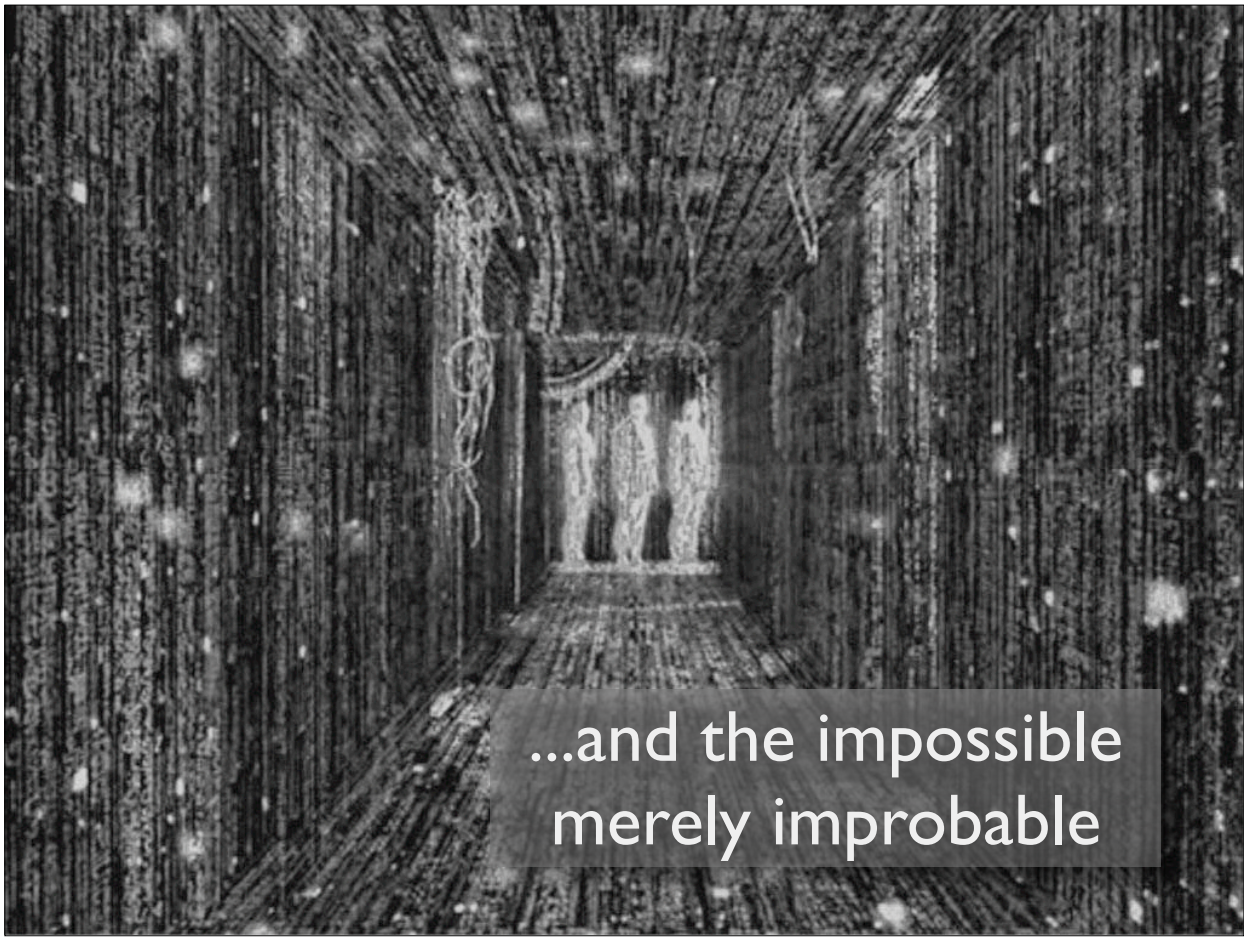
NF

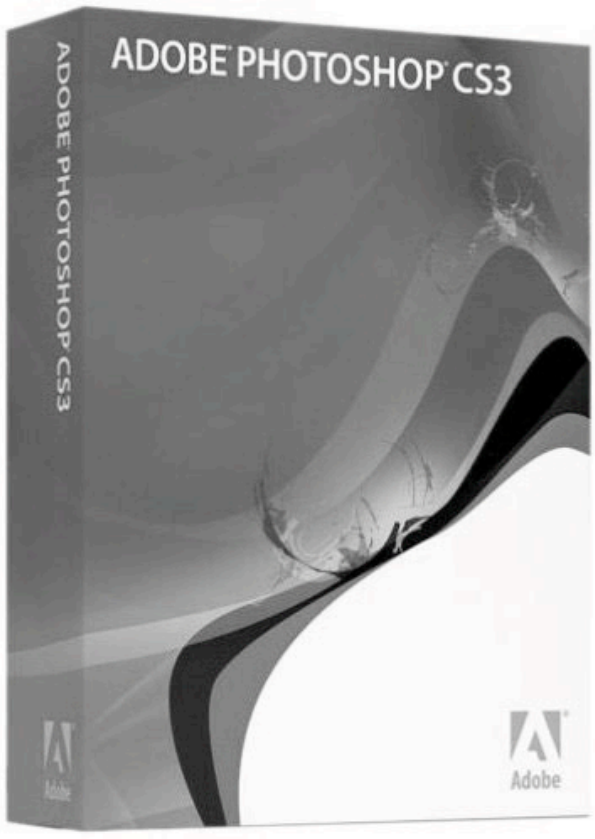
housekeeping

ask questions anytime

download slides from nealford.com →

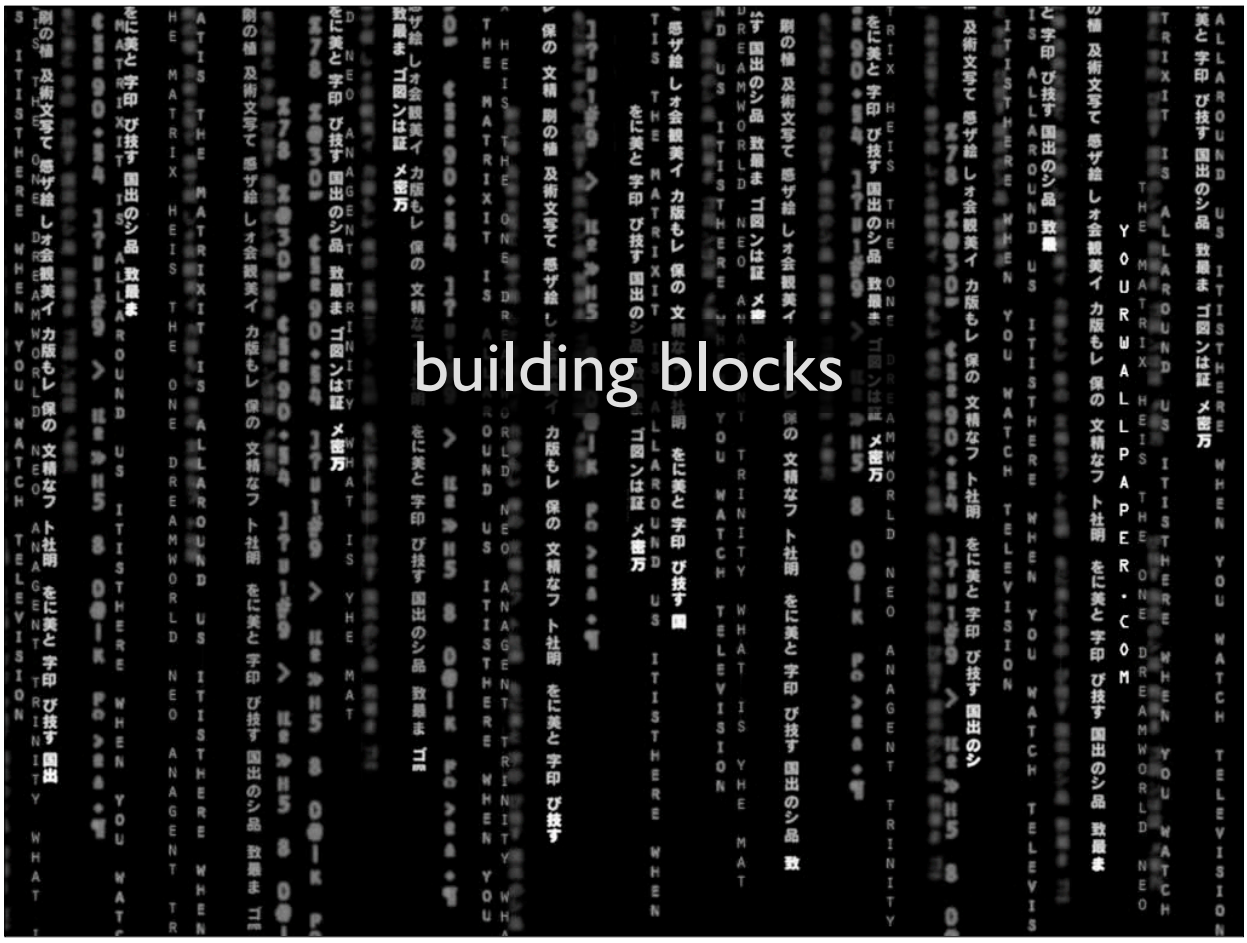
download samples from **`github.com/nealford`**

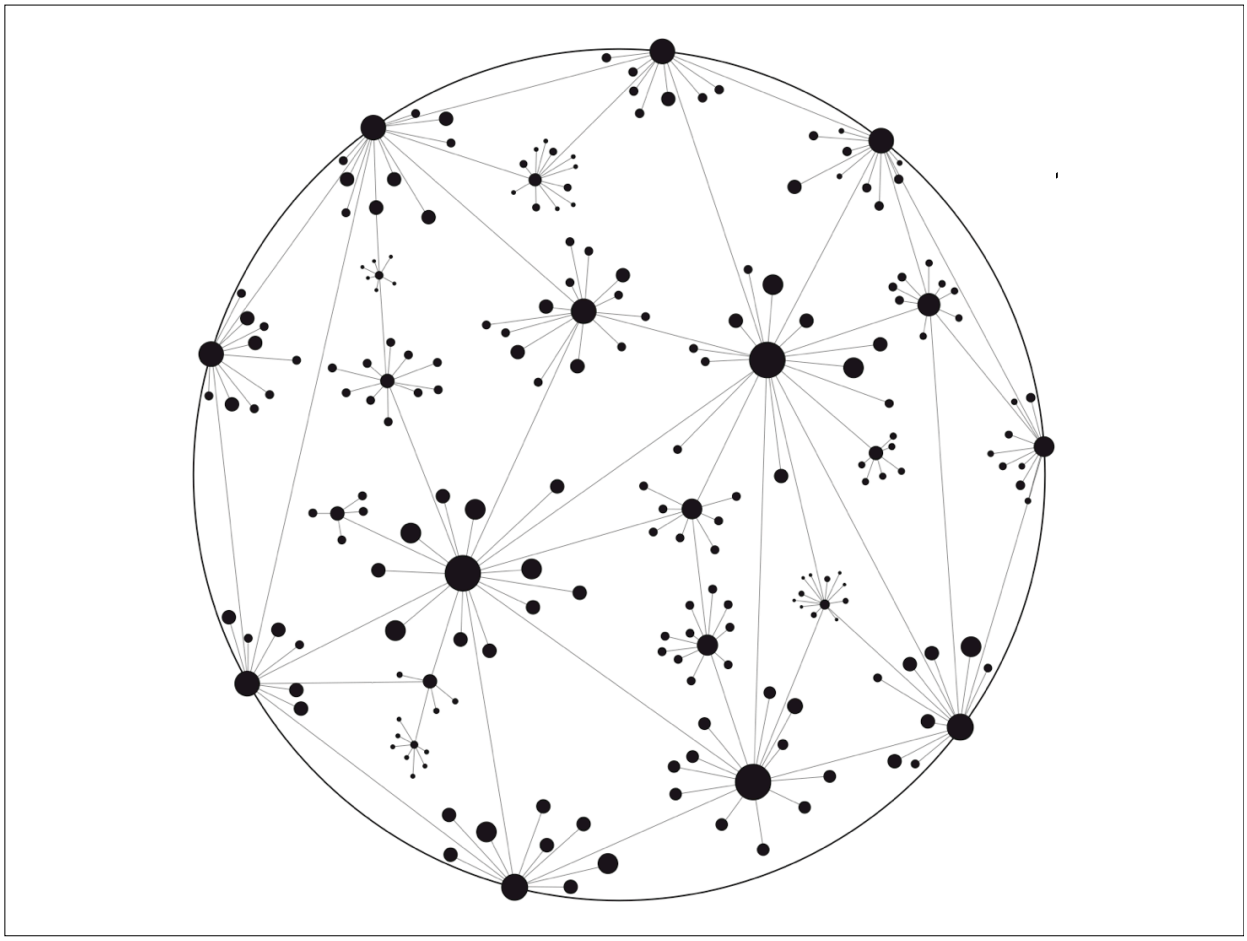

...and the impossible
merely improbable

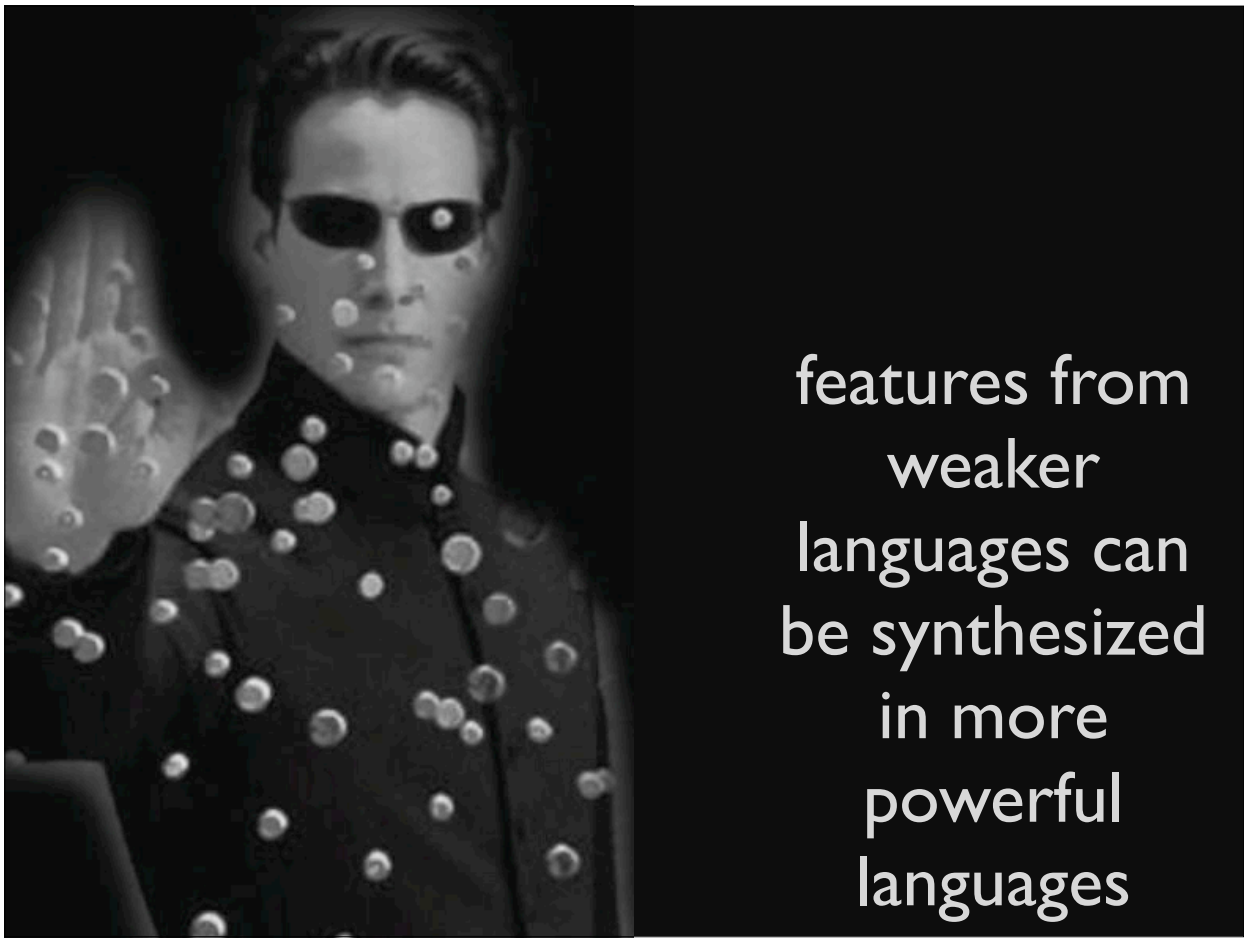

ADOBE PHOTOSHOP CS3
ADOBE PHOTOSHOP CS3
Adobe
Adobe

sapir-whorf hypothesis

language affects the capabilities of thoughts






features from
weaker
languages can
be synthesized
in more
powerful
languages

all computation in ruby

binding names to objects (assignment)

primitive control structures (if/else, while)

sending messages to objects

messages

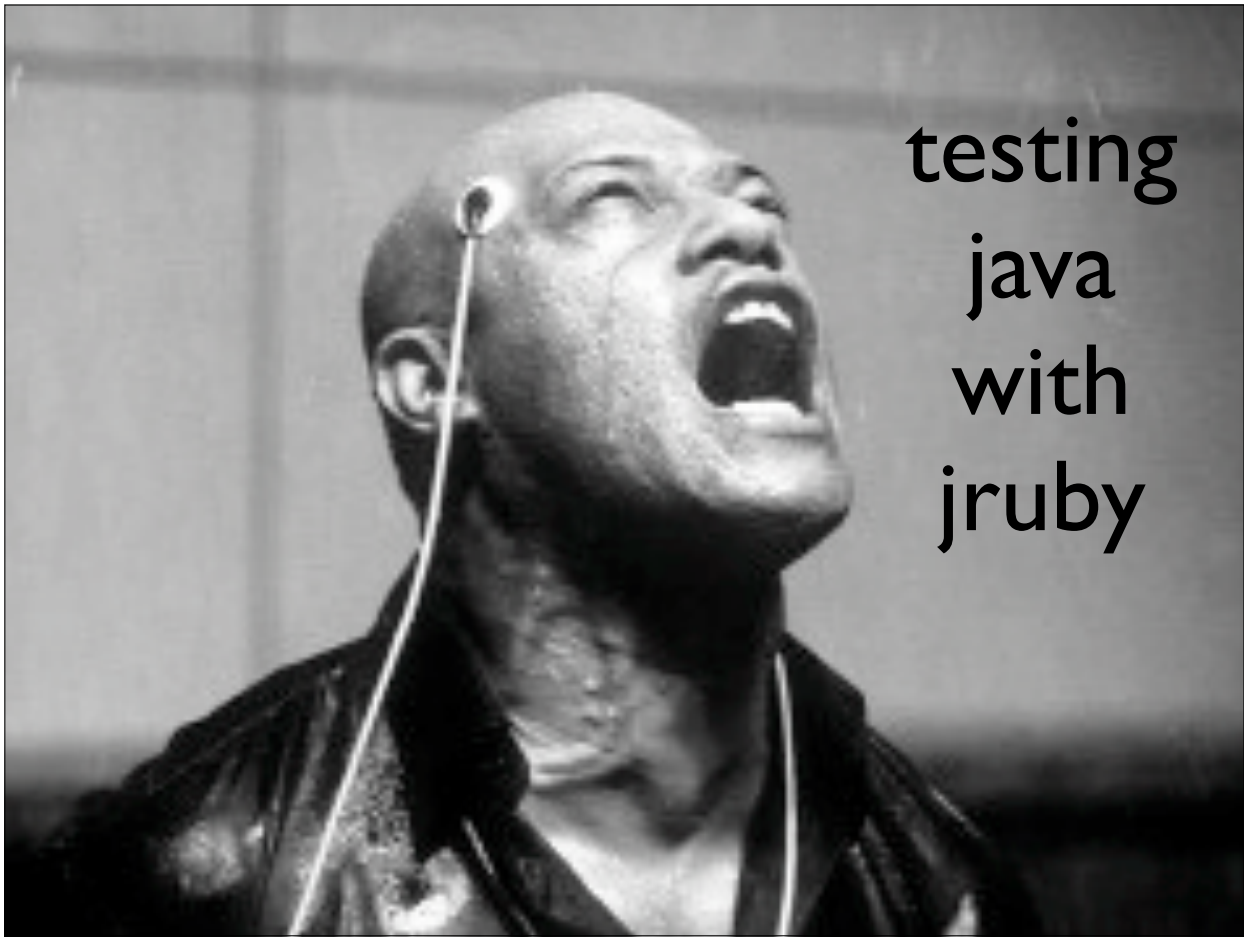
```
def test_messages_equal_method_calls
  tagline = "Unfortunately, no one can be told what the Matrix is."
  assert tagline[0..12].downcase == "unfortunately"
  assert tagline[0..12].send(:downcase) == "unfortunately"
  assert tagline[0..12].__send__(:downcase) == 'unfortunately'
  assert tagline[0..12].send("downcase".to_sym) == 'unfortunately'
end
```

construction isn't special

```
def test_construction
  a = Array.new
  assert a.kind_of? Array

  b = Array.send(:new)
  assert b.kind_of? Array
end
```

factory "design pattern"

```
def create_from_factory(factory)
  factory.new
end

def test_factory
  list = create_from_factory(Array)
  assert list.kind_of? Array

  hash = create_from_factory(Hash)
  assert hash.is_a? Hash
end
```



the java part

```
public interface Order {
    void fill(Warehouse warehouse);

    boolean isFilled();
}
public interface Warehouse {
    public void add(String item, int quantity);

    int getInventory(String product);

    boolean hasInventory(String product, int quantity);

    void remove(String product, int quantity);
}
```

testing `fill()`

```
public void fill(Warehouse warehouse) {
    if (warehouse.hasInventory(_product, _quantity)) {
        warehouse.remove(_product, _quantity);
        _filled = true;
    } else
        _filled = false;

}
```

jmock

```
@Test public void fillingRemovesInventoryIfInStock() {
    Order order = new OrderImpl(TALISKER, 50);
    final Warehouse warehouse = context.mock(Warehouse.class);

    context.checking(new Expectations() {{
        one (warehouse).hasInventory(TALISKER, 50); will(returnValue(true));
        one (warehouse).remove(TALISKER, 50);
    }});

    order.fill(warehouse);
    assertThat(order.isFilled(), is(true));
    context.assertIsSatisfied();
}
```

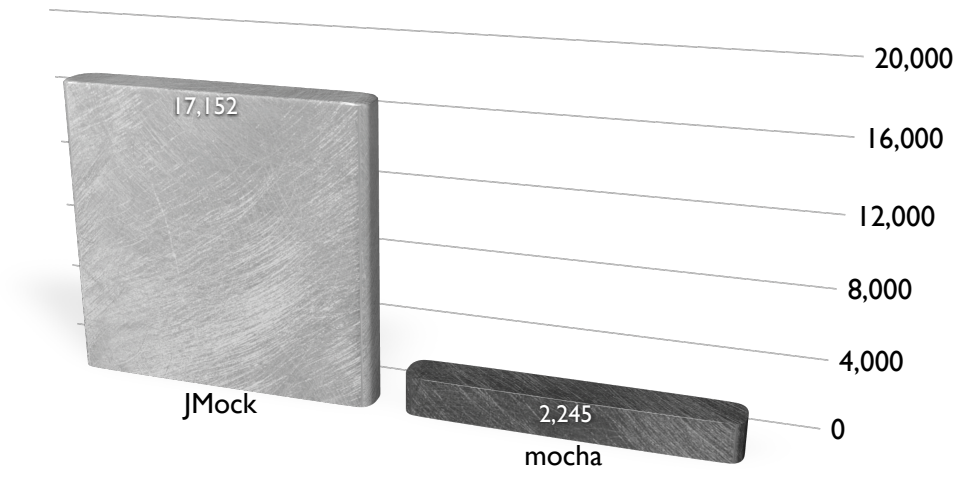
mocha

```
def test_filling_removes_inventory_if_in_stock
  order = OrderImpl.new(TALISKER, 50)
  warehouse = Warehouse.new
  warehouse.stubs(:hasInventory).
    with(TALISKER, 50).
    returns(true)
  warehouse.stubs(:remove).with(TALISKER, 50)

  order.fill(warehouse)
  assert order.is_filled
end
```

what does it take???

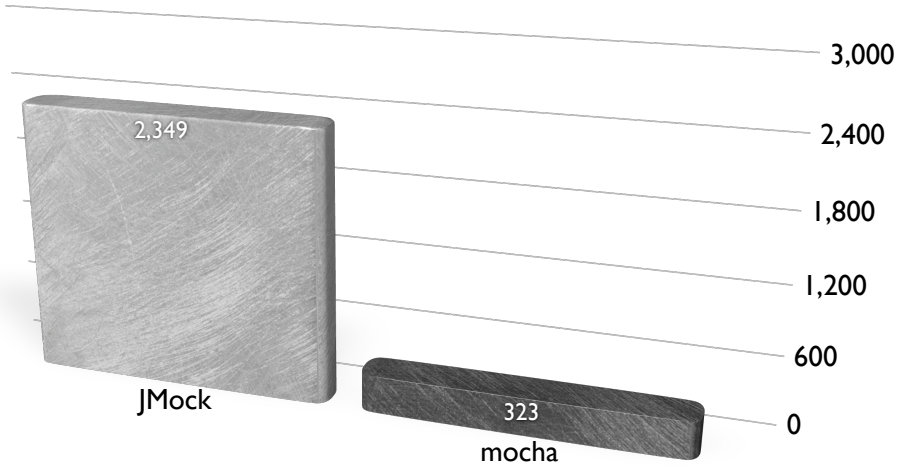
```
class Object

  def expects(symbol)
    method = stubba_method.new(stubba_object, symbol)
    $stubba.stub(method)
    mocha.expects(symbol, caller)
  end

  def stubs(symbol)
    method = stubba_method.new(stubba_object, symbol)
    $stubba.stub(method)
    mocha.stubs(symbol, caller)
  end

  def verify
    mocha.verify
  end

end
```

jmock vs mocha loc



jmock has 7.5 times as many lines of code

jmock vs mocha cc



jmock has 7.2 times the complexity of mocha

immutable string?

```
class ImmutableString
  attr_reader :value

  def initialize(s)
    @value = s.to_sym
  end

  def +(other)
    String.new(@value.to_s + other)
  end

  def <<(addition)
    return @value.to_s + addition
  end

  def to_s
    @value.to_s
  end

  def ==(other)
    @value.to_s == other
  end

end
```

```
def test_strings_have_the_same_object_id
  s1 = ImmutableString.new("foo")
  s2 = ImmutableString.new("foo")
  assert_equal s1.value.object_id, s2.value.object_id
end

def test_that_immutable_strings_are_immutable
  s1 = ImmutableString.new("foo")
  assert_raise NoMethodError do
    s1.value = "bar"
  end
end

def test_all_equals_methods_think_it_is_equal
  s1 = ImmutableString.new("foo")
  s2 = ImmutableString.new("foo")
  assert s1.value == s2.value
  assert s1.value.equal? s2.value
  assert s1.value.eql? s2.value
end
```

does it act like a string?

```
def test_that_it_acts_like_a_string
  s1 = ImmutableString.new("foo")
  s2 = "bar"
  assert "foobar" == s1 + s2
  assert s1 == "foo"
end

def test_chevron
  s1  = ImmutableString.new("foo")
  s1 << "bar"
  s2 = s1 << "bar"
  assert s2 == "foobar"
  assert s1 == "foo"
end
```

to be continued...

programmable programs

because ruby is interpreted...

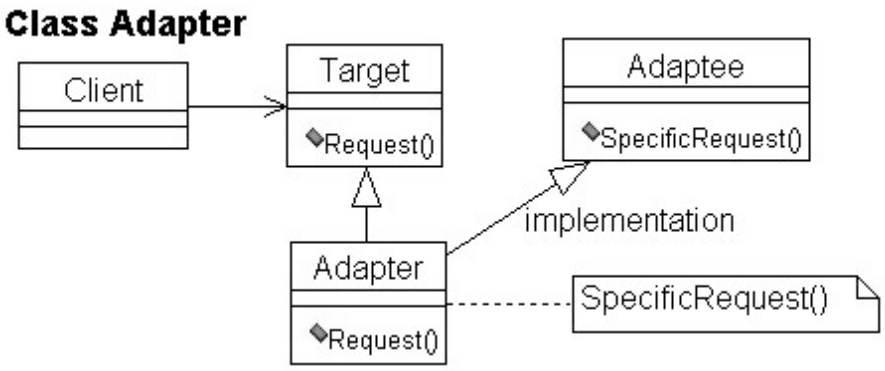
...and things happen as they are interpreted

you can do unexpected things...

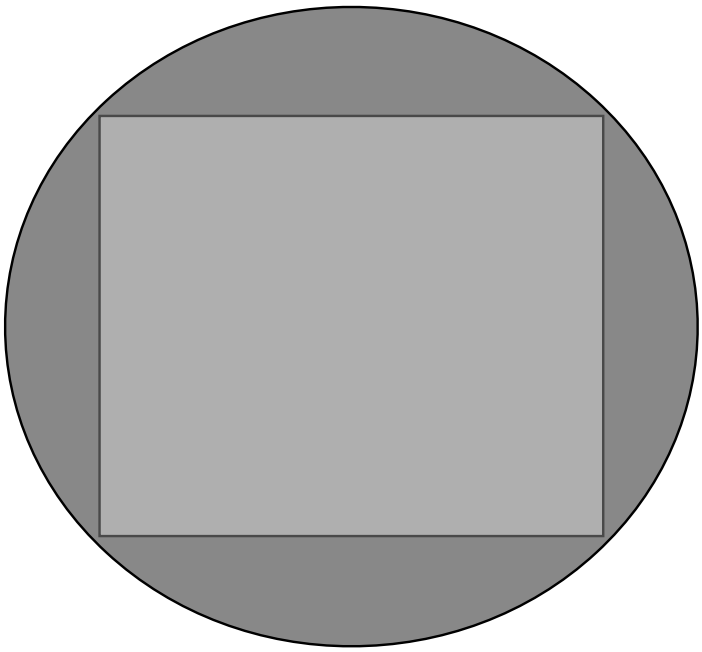
...like conditionally open classes

```ruby
def the_one?
  true
end

if the_one?
   class Spoon
     def bend
       "it is not the spoon that bends, it is only yourself."
     end
   end
else
   class Spoon
     def bend
       "do not try and bend the spoon"
     end
   end
end
```

```ruby
class TestOpenedString < Test::Unit::TestCase
  def test_spoon_bending
    assert_equal("it is not the spoon that bends, it is only yourself.",
      Spoon.new.bend)
  end

  def test_yourself_bending
    assert "do not try and bend the spoon" != Spoon.new.bend
  end

  def test_you_cannot_change_the_one
    def the_one?
      false
    end
    assert "do not try and bend the spoon" != Spoon.new.bend
  end
end
```



Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



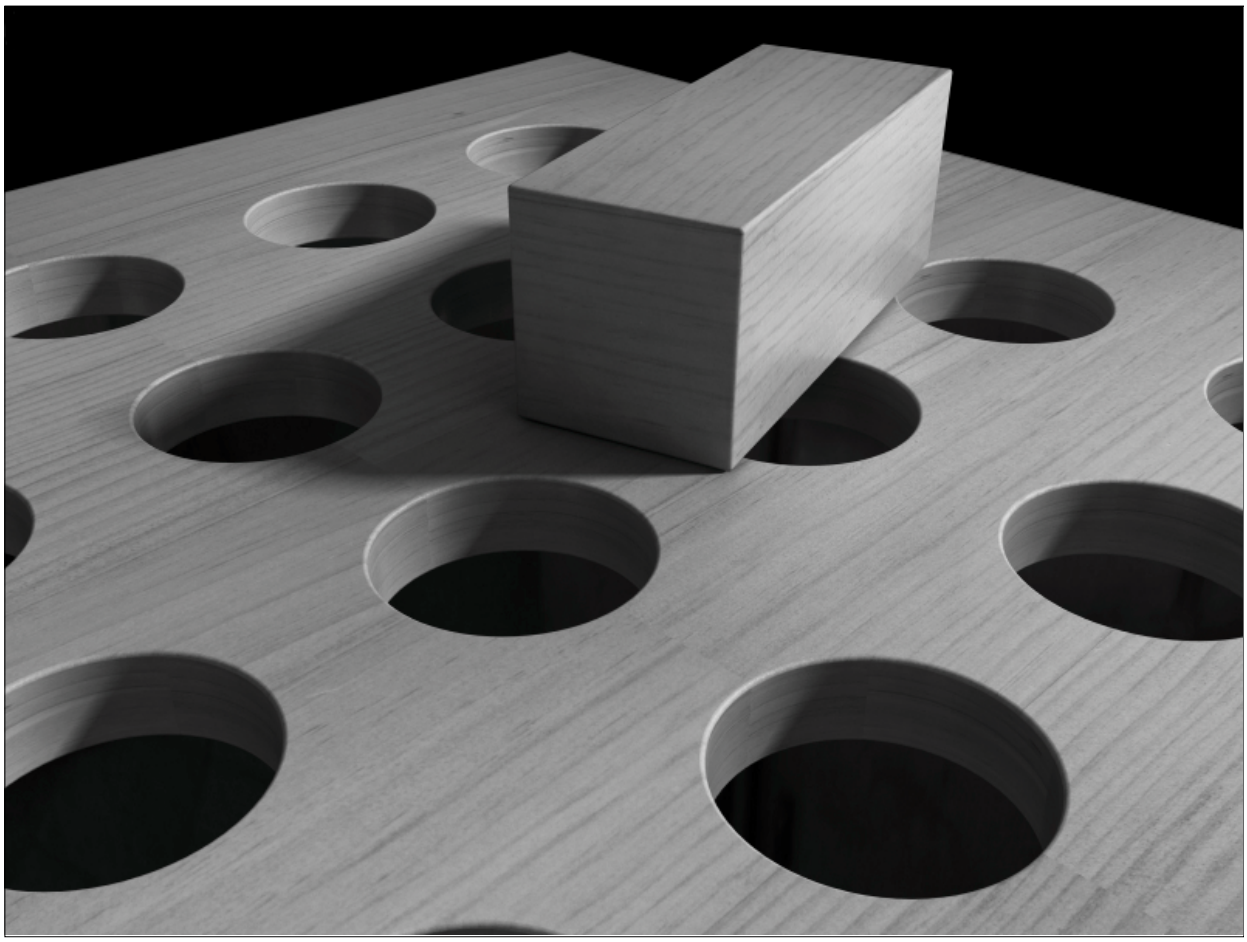
step 1:“normal” adaptor

```
class SquarePeg
    attr_reader :width

    def initialize(width)
        @width = width
    end
end

class RoundPeg
    attr_reader :radius

    def initialize(radius)
        @radius = radius
    end
end
```

```ruby
class RoundHole
    attr_reader :radius

    def initialize(r)
        @radius = r
    end

    def peg_fits?( peg )
        peg.radius <= radius
    end
end
```

```ruby
class SquarePegAdaptor
    def initialize(square_peg)
        @peg = square_peg
    end

    def radius
        Math.sqrt(((@peg.width/2) ** 2)*2)
    end
end
```

```
def test_pegs
  hole = RoundHole.new(4.0)
  4.upto(7) do |i|
      peg = SquarePegAdaptor.new(SquarePeg.new(i))
      if (i < 6)
        assert hole.peg_fits?(peg)
      else
        assert ! hole.peg_fits?(peg)
      end
  end
end
```



why bother with extra adaptor class?

```
class SquarePeg
    def radius
        Math.sqrt( ((width/2) ** 2) * 2 )
    end
end
```

what if open class added adaptor methods clash with existing methods?

```
class SquarePeg
  include InterfaceSwitching

  def radius
    @width
  end

  def_interface :square, :radius

  def radius
    Math.sqrt(((@width/2) ** 2) * 2)
  end

  def_interface :holes, :radius

  def initialize(width)
    set_interface :square
    @width = width
  end
end
```

```
def test_pegs_switching
  hole = RoundHole.new( 4.0 )
  4.upto(7) do |i|
    peg = SquarePeg.new(i)
    peg.with_interface(:holes) do
      if (i < 6)
        assert hole.peg_fits?(peg)
      else
        assert ! hole.peg_fits?(peg)
      end
    end
  end
end
```

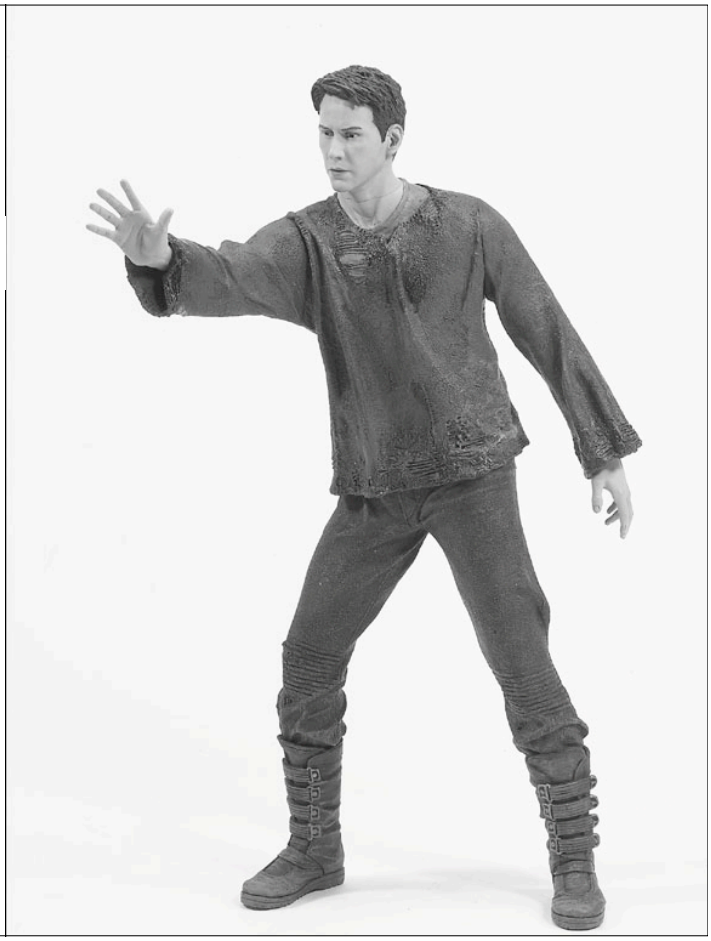
interface helper

```
class Class
  def def_interface(interface, *syms)
    @__interface__ ||= {}
    a = (@__interface__[interface] ||= [])
    syms.each do |s|
      a << s unless a.include? s
      alias_method "__#{s}_#{interface}__".intern, s
      remove_method s
    end
  end
end
```

```
module InterfaceSwitching
  def set_interface(interface)
    unless self.class.instance_eval{ @__interface__[interface] }
      raise "Interface for #{self.inspect} not understood."
    end
    i_hash = self.class.instance_eval "@__interface__[interface]"
    i_hash.each do |meth|
      class << self; self end.class_eval <<-EOF
        def #{meth}(*args,&block)
                send(:__#{meth}_#{interface}__, *args, &block)
        end
      EOF
    end
    @__interface__ = interface
  end

  def with_interface(interface)
    oldinterface = @__interface__
    set_interface(interface)
    begin
      yield self
    ensure
      set_interface(oldinterface)
    end
  end
end
```

compilation is premature optimization

modules



block syntax

```
def use_block flag
  yield if flag
end

use_block(1 == 1) { puts "What is the Matrix?"}

use_block(1 == 2) do
  puts "The answer is out there, Neo"
end
```

`quantifier` module

```
module Quantifier
  def any?
    each { |x| return true if yield x }
    false
  end

  def all?
    each { |x| return false if not yield x }
    true
  end
end
```

make arrays quantifiable

```
class Array
   include Quantifier
end
```

1. mixin with open class

```
list = Array.new
list.extend Quantifier
```

2. extending a single instance

```ruby
class TestQuantifier < Test::Unit::TestCase

  def setup
    @list = []
    1.upto(20) do |i|
      @list << i
    end
  end

  def test_any
    assert @list.any? {|x| x > 5 }
    assert ! @list.any? {|x| x > 20 }
  end

  def test_all
    assert @list.all? { |x| x < 50 }
    assert !@list.all? { |x| x < 10 }
  end

end
```

```ruby
class TestQuantifierWithExtension < Test::Unit::TestCase

  def setup
    @list = []
    @list.extend(Quantifier)
    1.upto(20) do |i|
      @list << i
    end
  end

  def test_any
    assert @list.any? {|x| x > 5 }
    assert ! @list.any? {|x| x > 20 }
  end

  def test_all
    assert @list.all? { |x| x < 50 }
    assert !@list.all? { |x| x < 10 }
  end

end
```

what if we wanted to count everything we quantified?

```
module Quantifier
  @@quantified_count = 0

  def Quantifier.append_features(targetClass)
    def targetClass.quantified_count
      @@quantified_count
    end
    super
  end

  def any?
    each do |x|
      @@quantified_count += 1
      return true if yield x
    end
    false
  end

  def all?
    each do |x|
      @@quantified_count += 1
      return false if not yield x
    end
    true
  end
end
```

```
class TestQuantifierWithExtension < Test::Unit::TestCase

  def setup
    @list = []
    @list.extend(Quantifier)
    1.upto(20) do |i|
      @list << i
    end
  end

  def test_any
    assert @list.any? {|x| x > 5 }
    assert ! @list.any? {|x| x > 20 }
  end

  def test_all
    assert @list.all? { |x| x < 50 }
    assert !@list.all? { |x| x < 10 }
  end

end
```

sort on any field of a class

the ruby way

```
class Array
   def sort_by(attribute)
     sort {|x, y| x.send(attribute) <=> y.send(attribute) }
   end
end

class Person
  attr_reader :name, :age, :height

  def initialize(name, age, height)
    @name, @age, @height = name, age, height
  end

  def to_s
    "Name: #{@name} is #{@age} years old and #{height} tall."
  end
end
```

```ruby
people = []
people << Person.new("Neo", 30, 6)
people << Person.new("Trinity", 29, 5.6)
people << Person.new("Morpheus", 40, 5.9)

by_name = people.sort_by :name
by_name.each {|p| puts p.name}
people.sort_by(:age).each {|p| puts p.age}
```

```
public Comparator<Employee> getComparatorFor(final String field) {
    return new Comparator<Employee> () {
        public int compare(Employee o1, Employee o2) {
            Object field1, field2;
            try {
                field1 = method.invoke(o1, null);
                field2 = method.invoke(o2, null);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
            return ((Comparable) field1).compareTo(field2);
        }

    };
}
```

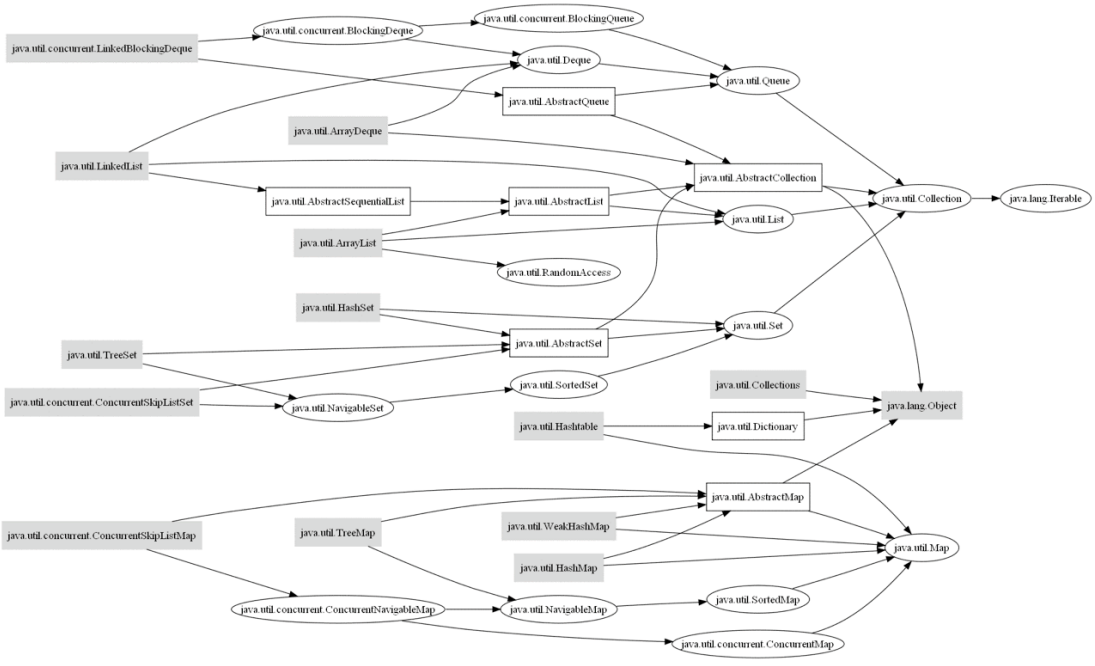
```ruby
class Array
  def sort_by_attribute(sym)
     sort {|x,y| x.send(sym) <=> y.send(sym) }
  end
end
```

interfaces in ruby?

```
module Iterator
  def initialize
    %w(hasNext next).each do |m|
      unless self.class.public_method_defined? m
        raise NoMethodError
      end
    end
  end
end
```

```
class TestInterfaceDemo < Test::Unit::TestCase

  class Foo; include Iterator; end

  class Foo2; include Iterator; def hasNext; end; end

  class Foo3; include Iterator; def hasNext; end; def next; end
  end

  def test_methods_exist_when_imposed
    assert_raise(NoMethodError) {
      Foo.new
    }
  end

  def test_interface_imposition_fails_when_only_1_method_present
    assert_raise(NoMethodError) {
      Foo2.new
    }
  end

  def test_interface_works_when_interfaces_implemented
    f = Foo3.new
    assert f.class.public_method_defined? :hasNext
    assert f.class.public_method_defined? :next
  end

end
```

java's collection package



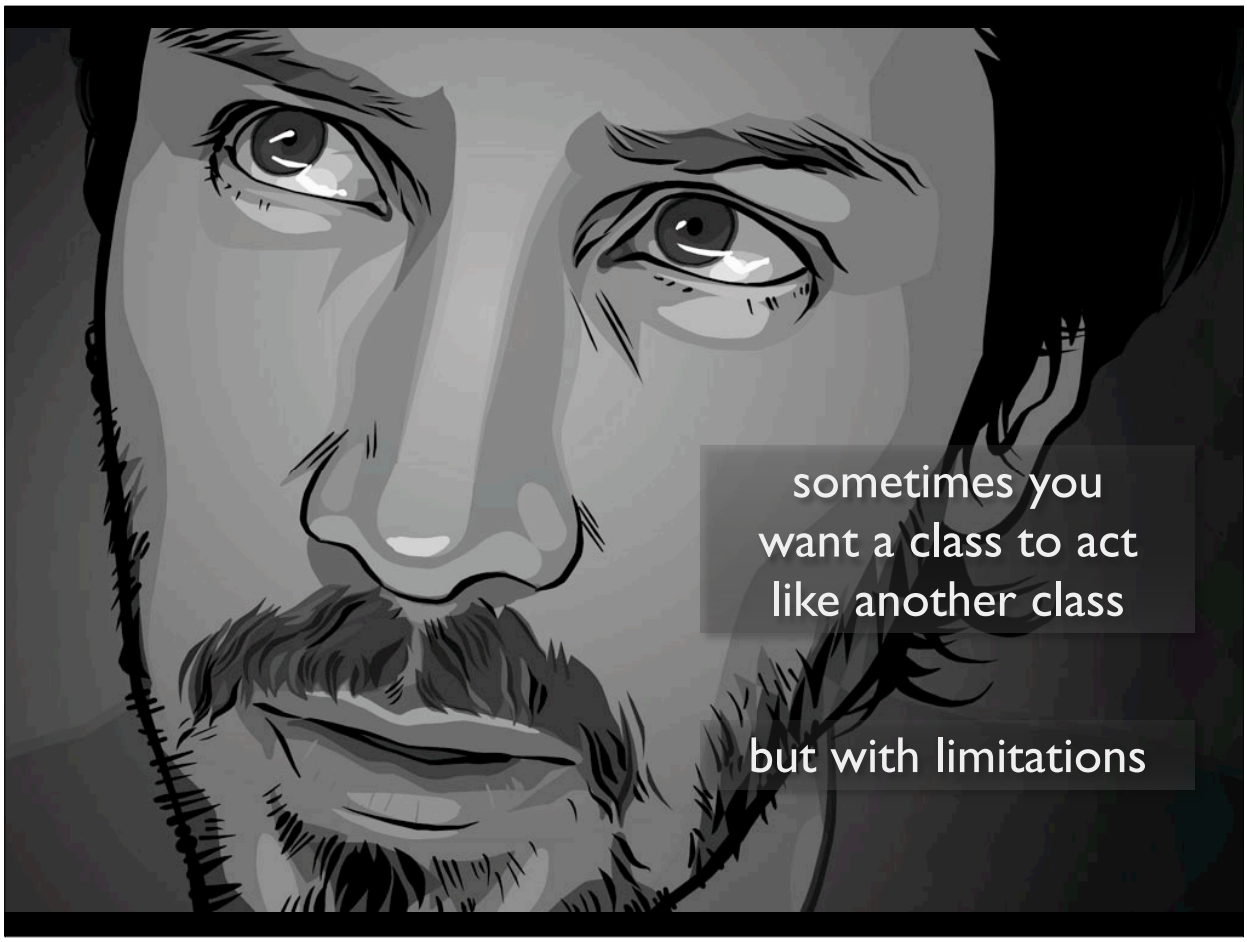
ruby's collections

Array

Set

"humane interface"

Hash



queue class

```
require 'delegate'

class DelegateQueue < DelegateClass(Array)
  def initialize(arg=[])
    super(arg)
  end

  alias_method :enqueue, :push
  alias_method :dequeue, :shift
end
```

```
def setup
  @q = DelegateQueue.new
  @q.enqueue "one"
  @q.enqueue "two"
end

def test_queuing
  e = @q.dequeue
  assert_equal "one", e
end
```

```
def test_non_delegated_methods
  @q = DelegateQueue.new
  @q.enqueue "one"
  @q.enqueue "two"
  assert_equal 2, @q.size
  e = @q.dequeue
  assert_equal 1, @q.size
  assert_equal e, "one"
end
```

a delegate is just a wrapper around another class

forwarding

```
require 'forwardable'

class FQueue
  extend Forwardable

  def initialize(obj=[])
    @queue = obj
  end

  def_delegator :@queue, :push, :enqueue
  def_delegator :@queue, :shift, :dequeue
  def_delegators :@queue, :clear,
      :empty?, :length, :size, :<<
end
```

```
def test_queue
  e = @q.dequeue
  assert_equal "one", e
end

def test_delegated_methods
  @q.enqueue "three"
  assert_equal 3, @q.size
  e = @q.dequeue
  assert_equal 2, @q.size
  assert_equal "one", e
  @q.clear
  assert_equal 0, @q.size
  assert @q.empty?
  assert_equal 0, @q.length
  @q << "new"
  assert_equal 1, @q.length
end
```

non-delegating methods

```
def test_non_delegated_methods
  assert_raise(NoMethodError) { @q.pop }
end

def test_delegating_to_array
  arr = Array.new
  q = FQueue.new arr
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end
```

```
def test_delegating_to_a_queue
  a = Queue.new
  q = FQueue.new a
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end

def test_delgating_to_a_sized_queue
  a = SizedQueue.new(12)
  q = FQueue.new a
  q.enqueue "one"
  assert_equal 1, q.size
  assert_equal "one", q.dequeue
end
```

any duck

```
class FQueue
  extend Forwardable

  def initialize(obj=[])
    @queue = obj
  end

  def_delegator :@queue, :push, :enqueue
  def_delegator :@queue, :shift, :dequeue
  def_delegators :@queue, :clear,
      :empty?, :length, :size, :<<
end
```

things gone missing

when you call a method or reference a constant that isn't around

ruby handles it with a **`missing`** method

`const_missing`

`method_missing`

handle it any way you like

command wrapper

```
class CommandWrapper
  def method_missing(method, *args)
    system(method.to_s, *args)
  end
end
```

```
class TestCommandWrapper < Test::Unit::TestCase

  def setup
    @cw = CommandWrapper.new
  end

  def test_current_date
    expected = system('date')
    assert_equal expected, @cw.date
  end

  def test_ls
    expected = system('ls')
    assert_equal expected, @cw.ls
  end
end
```

recorder

```
class Recorder
  def initialize
    @messages = []
  end

  def method_missing(method, *args, &block)
    @messages << [method, args, block]
  end

  def play_back_to(obj)
    @messages.each do |method, args, block|
      obj.send(method, *args, &block)
    end
  end
end
```

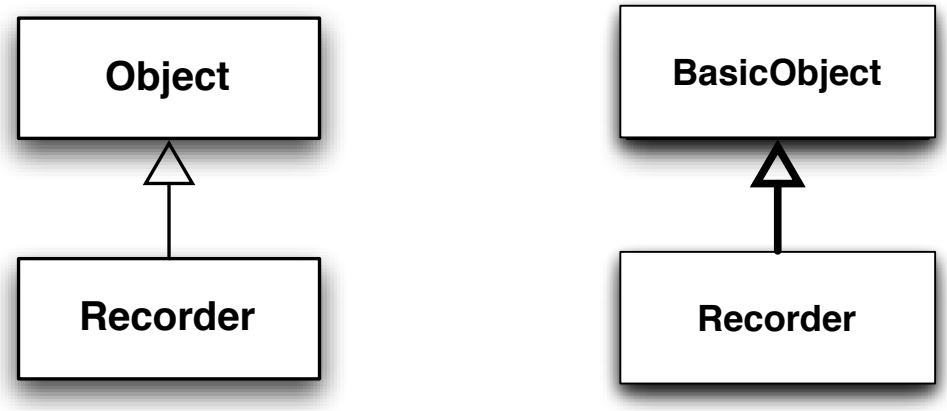
```ruby
def test_recorder
  r = Recorder.new
  r.sub!(/Java/) { "Ruby" }
  r.upcase!
  r[11, 5] = "Universe"
  r << "!"

  s = "Hello Java World"
  r.play_back_to(s)
  assert_equal "HELLO RUBY Universe!", s
end
```

but what about this?

```ruby
def test_recorder_fails_when_existing_methods_called
  r = Recorder.new
  r.downcase!
  r.freeze

  s = "Hello Ruby"
  r.play_back_to s
  assert_equal("hello ruby", s)
  assert_equal(s.upcase!, "HELLO RUBY")
end
```

should fail because ***s*** should be frozen

1.8 Jim Weirich's BlankSlate



1.9 BasicObject

```
class Recorder2 < BlankSlate
  def initialize
    @messages = []
  end

  def method_missing(method, *args, &block)
    @messages << [method, args, block]
  end

  def play_back_to(obj)
    @messages.each do |method, args, block|
      obj.send(method, *args, &block)
    end
  end
end
```

```ruby
def test_recorder_works_with_blankslate
  r = Recorder2.new
  r.downcase!
  r.freeze

  s = "Hello Ruby"
  r.play_back_to s
  assert_equal("hello ruby", s)
  assert_raise(TypeError) {
    s.upcase!
  }
end
```

method magic

runtime access to methods

create methods with **`define_method`**

get rid of methods

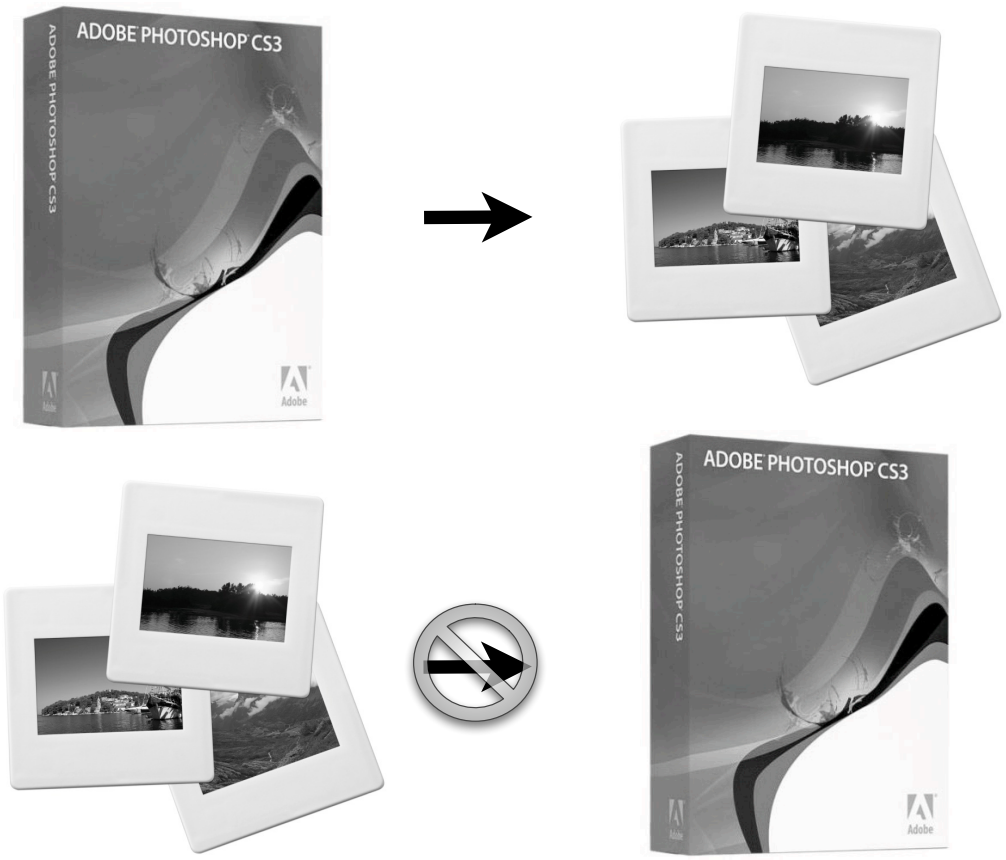
`remove_method` - from the current class

`undef_method` - from the entire hierarchy!

immutable string, take 2

```
class String
  instance_methods.each do |m|
    undef_method m.to_sym if m =~ /.*!$/
  end
end
```

```
class TestUnupdateableString < Test::Unit::TestCase

  def test_other_methods
    s1 = String.new "foo"
    assert_raise NoMethodError do
      s1.downcase!
    end

    assert_raise NoMethodError do
      s1.capitalize!
    end
  end

  def test_that_methods_still_work
    s1 = "foo"
    s2 = s1 + 'bar'
    assert "foobar" == s2
  end
end
```



adding `final`

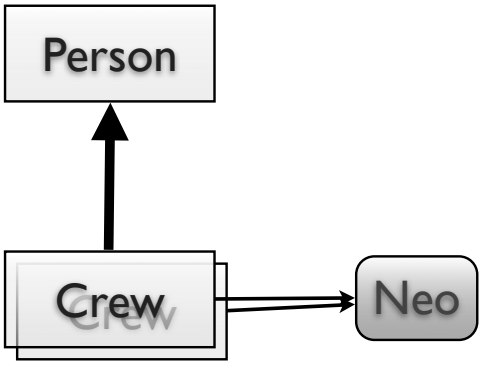
```
module Final
  def self.included(c)
    c.instance_eval do
      def inherited(sub)
        raise Exception,
            "Attempt to create subclass #{sub} "
            "of Final class #{self}"
      end
    end
  end
end
```

```
class P; include Final; end

class C < P; end
```

singleton method

eigenclass

eigenclass



the ability to add methods
to object instances

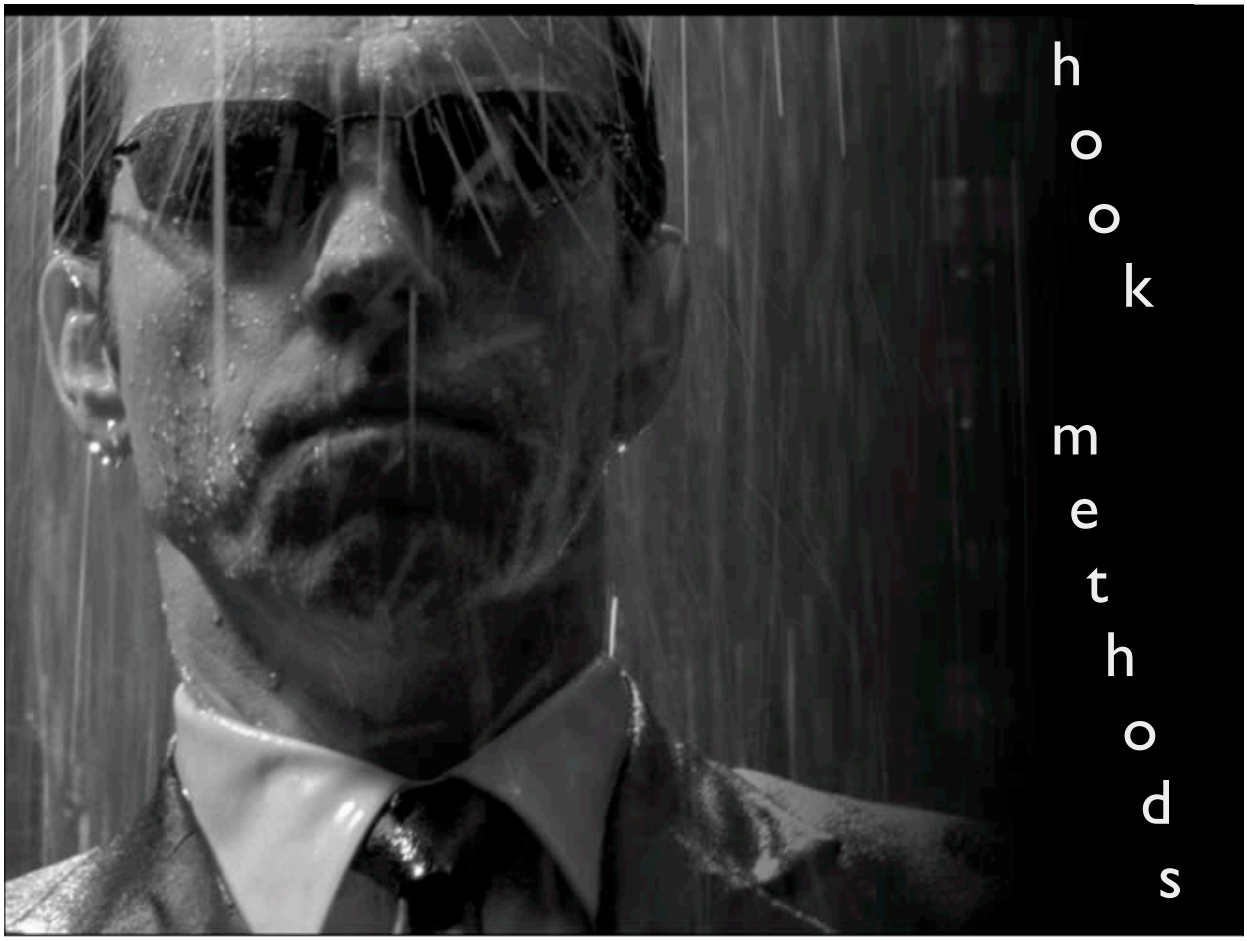
adding methods via proxies

```
require "java"

include_class "java.util.ArrayList"

class ArrayList
  def first
    size == 0 ? nil : get(0)
  end
end
```

```
class TestArrayListProxy < Test::Unit::TestCase
  def setup
    @list = ArrayList.new
    @list << 'Red' << 'Green' << 'Blue'
    def @list.last
      size == 0 ? nil : get(size - 1)
    end
  end

  def test_first
    assert_equal "Red", @list.first
  end

  def test_last
    assert_equal "Blue", @list.last
  end
```

metaclass/ eigenclass

```
class Object
   def eigenclass
     class << self
       self
     end
   end
end
```



logging

```
require 'singleton'

class Log
  include Singleton
  def write(msg)
    puts msg
  end
end


class OldFashioned
  def some_method
    Log.instance.write("starting method 'some_method'")
    puts "do something important"
    Log.instance.write("ending method 'some_method'")
  end
end
```

```ruby
module Aop
  def Aop.included(into)
    into.instance_methods(false).each { |m| Aop.hook_method(into, m) }

    def into.method_added(meth)
      unless @adding
        @adding = true
        Aop.hook_method(self, meth)
        @adding = false
      end
    end
  end

  def Aop.hook_method(klass, meth)
    klass.class_eval do
      if meth.to_s =~ /^persist_.*/
        alias_method "old_#{meth}", "#{meth}"
        define_method(meth) do |*args|
          Log.instance.write("calling method #{meth}")
          self.send("old_#{meth}",*args)
          Log.instance.write("call finished for #{meth}")
        end
      end
    end
  end
end
```

is monkey patching evil?

aspect nomenclature

join point

points of program execution where new behavior might be inserted.

pointcut

sets of *join points* with a similar "theme"

advice

code invoked before, after, or around a *join point*

aspect oriented ruby

interception

interjection of advice, at least around methods

introduction

enhancing with new (orthogonal!) state & behavior

inspection

access to meta-information that may be exploited by pointcuts or advice

modularization

encapsulate as aspects

aop: interception

```
class Customer
  def update
    save
  end
end

class Customer
  alias :old_update, :update
  def update
    Log.instance.write("Saving")
    old_update
  end
end
```

alias name clashes

new method available

better interception

capture the target method as an unbound method

bind it to the current object

call it explicitly

```
class Customer
  old_update = self.instance_method(:update)
  def save
     Log.instance.write("Saving")
     old_update.bind(self).call
  end
end
```

alias_method_chain

```
module Layout #:nodoc:
  def self.included(base)
    base.extend(ClassMethods)
    base.class_eval do
      alias_method :render_with_no_layout, :render
      alias_method :render, :render_with_a_layout
      # ... etc


alias_method_chain :render, :layout
```

alias_method_chain

```
class Module
  # Encapsulates the common pattern of:
  #
  #   alias_method :foo_without_feature, :foo
  #   alias_method :foo, :foo_with_feature
  #
  # With this, you simply do:
  #
  #   alias_method_chain :foo, :feature
  #
  # And both aliases are set up for you.
  def alias_method_chain(target, feature)
    alias_method "#{target}_without_#{feature}", target
    alias_method target, "#{target}_with_#{feature}"
  end
end
```

aop: introductions

add a new method to a class

add a new method to an instance of a class (via the eigenclass)

aop: inspections

```
i=42
s="whoa"
local_variables
global_variables
s.class
s.display
s.inspect
s.instance_variables
s.methods
s.private_methods
s.protected_methods
s.public_methods
s.singleton_methods
s.method(:size).arity
s.method(:replace).arity
. . .
```

aop: modularization

```
class Person
  attr_accessor :name

  def initialize name
    @name = name
  end
end

class EntityObserver
  def receive_update subject
   puts "adding new name: #{subject.name}"
  end
end
```

```
module Subject
  def add_observer observer
    raise "Observer must respond to receive_update" unless
      observer.respond_to? :receive_update
    @observers ||= []
    @observers.push observer
  end

  def notify subject
    @observers.each { |o| o.receive_update subject }
  end
end

class Person
  include Subject
  old_name = self.instance_method(:name=)

  define_method(:name=) do |new_name|
    old_name.bind(self).call(new_name)
    notify self
  end
end
```

aop: modularization

```
neo = Person.new "neo"
morpheus = Person.new "morpheus"
neo.add_observer EntityObserver.new
neo.add_observer EntityObserver.new
morpheus.add_observer EntityObserver.new
neo.name = "the one"
morpheus.name = "the prophet"
```

aquarium

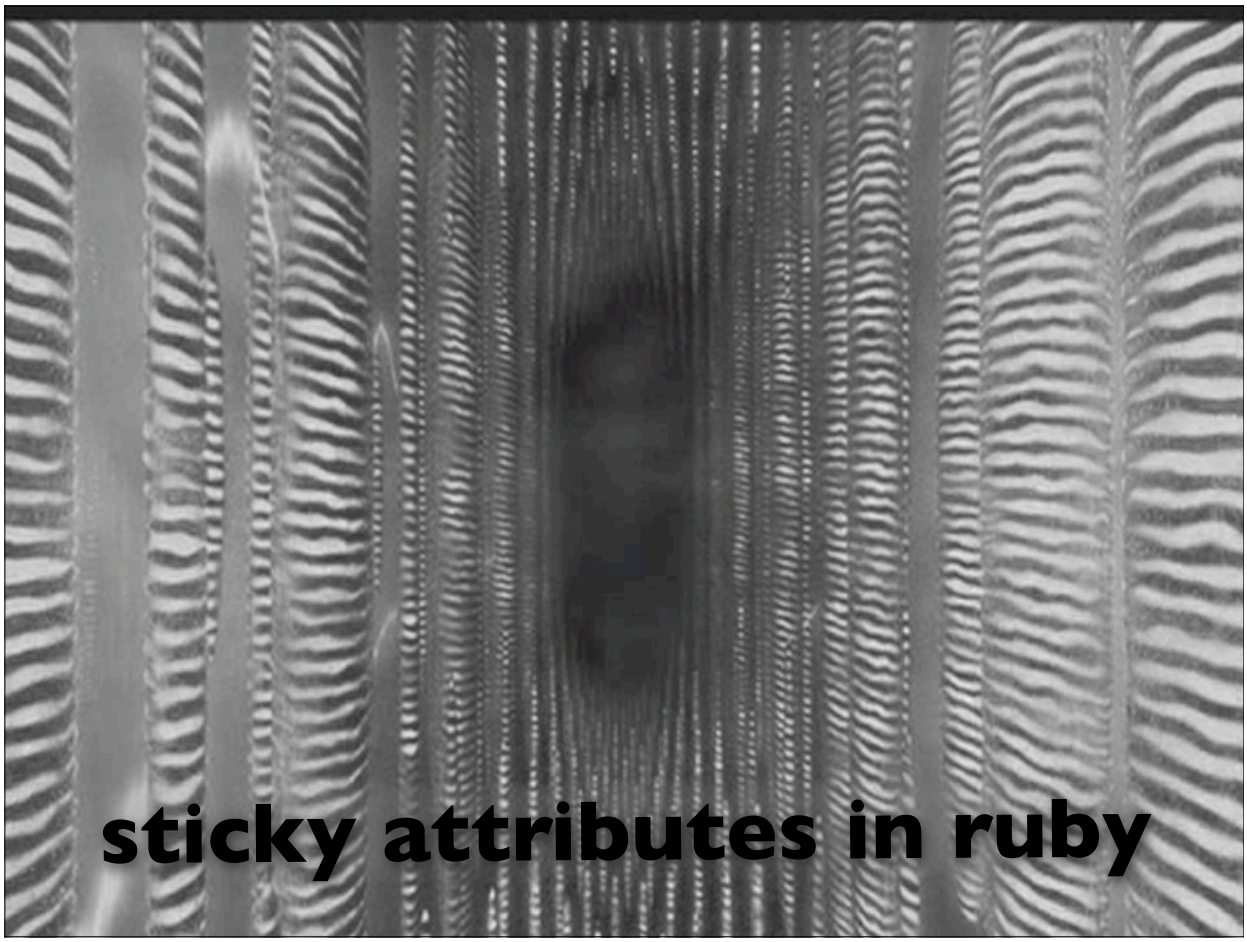
trace all invocations of the public instance methods in all classes whose names end with "Service"

```
class ServiceTracer
    include Aquarium::Aspects::DSL::AspectDSL
    before :calls_to => :all_methods,
           :in_types => /Service$/ do |join_point, object, *args|
      log "Entering: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
    end
    after :calls_to => :all_methods,
          :in_types => /Service$/ do |join_point, object, *args|
      log "Leaving: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
    end
end
```

aquarium

using *around* advice

```
class ServiceTracer
    include Aquarium::Aspects::DSL::AspectsDSL
    around :calls_to => :all_methods,
           :in_types => /Service$/ do |join_point, object, *args|
      log "Entering: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
      result = join_point.proceed
      log "Leaving: #{join_point.target_type.name}#" +
          "#{join_point.method_name}: object = #{object}, args = #{args}"
      result  # block needs to return the result of the "proceed"!
    end
end
```



limiting testing

```
require 'test/unit'
class CalculatorTest<Test::Unit::TestCase

  def test_some_complex_calculation
    assert_equal 2, Calculator.new(4).complex_calculation
  end

end
```

conditional method definition

```
class CalculatorTest<Test::Unit::TestCase

  if ENV["BUILD"] == "ACCEPTANCE"

    def test_some_complex_calculation
      assert_equal 2, Calculator.new(4).complex_calculation
    end

  end

end
```

attribute

```
class CalculatorTest<Test::Unit::TestCase
  extend TestDirectives

  acceptance_only
  def test_some_complex_calculation
    assert_equal 2, Calculator.new(4).complex_calculation
  end

end
```

using hook methods

```
module TestDirectives

  def acceptance_only
    @acceptance_build = ENV["BUILD"] == "ACCEPTANCE"
  end

  def method_added(method_name)
    remove_method(method_name) unless @acceptance_build
    @acceptance_build = false
  end

end
```

delineating blocks

```
class CalculatorTest<Test::Unit::TestCase
  extend TestDirectives

  acceptance_only do

    def test_some_complex_calculation
      assert_equal 2, Calculator.new(4).complex_calculation
    end

  end

end
```

building block container

```
module TestDirectives

  def acceptance_only &block
    block.call if ENV["BUILD"] == "ACCEPTANCE"
  end

end
```

named blocks

```
class CalculatorTest<Test::Unit::TestCase
  extend TestDirectives

  acceptance_only :test_some_complex_calculation do

    assert_equal 2, Calculator.new(4).complex_calculation

  end

end
```

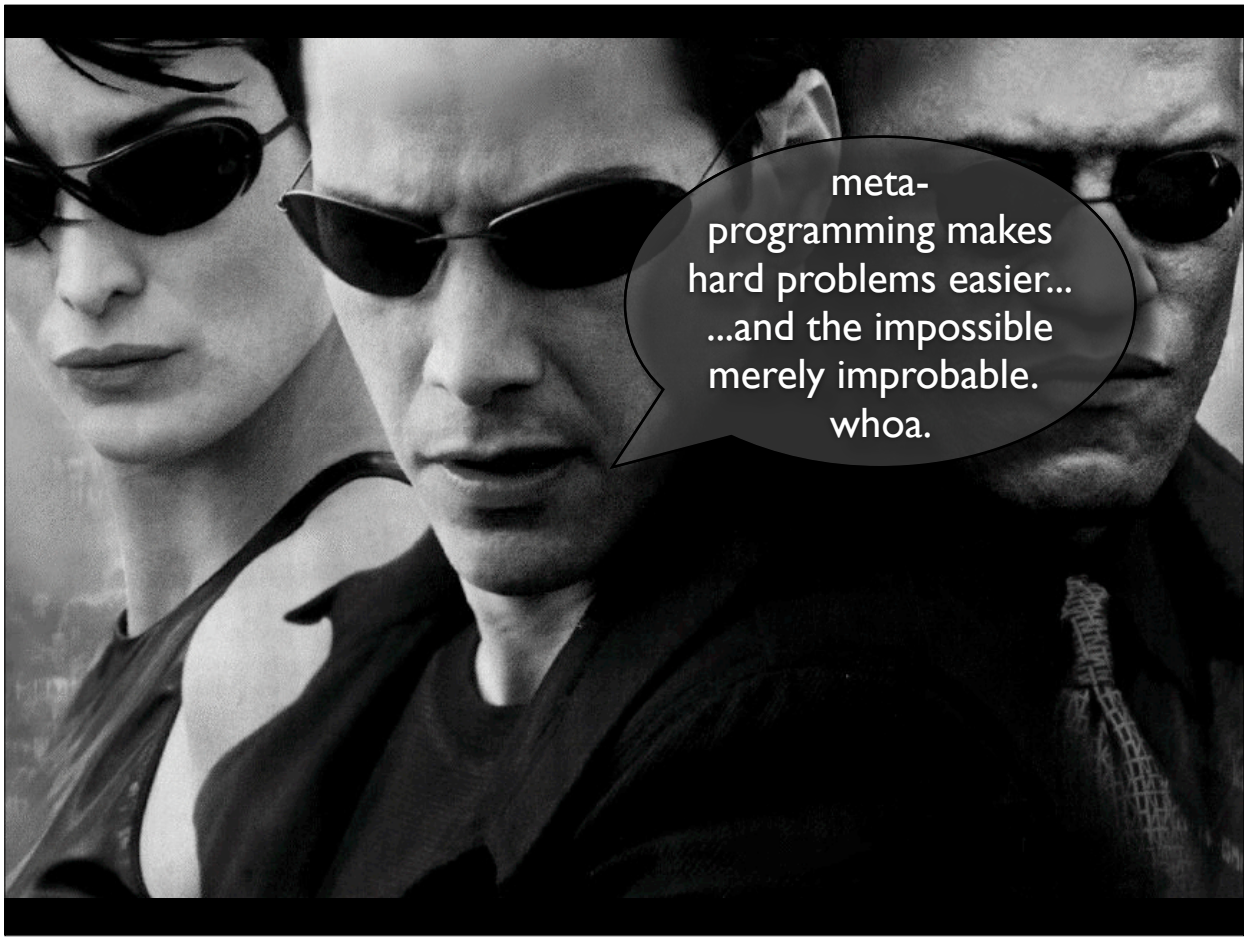
implementing named blocks

```
module TestDirectives

  def acceptance_only(method_name, &method_body)
    if ENV["BUILD"] == "ACCEPTANCE"
      define_method method_name, method_body
    end
  end

end
```

attributes for cross-cutting concerns

```
class Approval
  extend Loggable

  logged
  def decline(approver, comment)
    #implementation
  end

end
```

```
module Loggable
  def logged
    @logged = true
  end

  def method_added(method_name)
    logged_method = @logged
    @logged = false

    if logged_method
      original_method = :"unlogged_#{method_name.to_s}"
      alias_method original_method, method_name

      define_method(method_name) do |*args|
        arg_string = args.collect{ |arg| arg.inspect + " " } unless args.empty?
        log_message = "called #{method_name}"
        log_message << " with #{arg_string}" if arg_string
        Logger.log log_message
        self.send(original_method, *args)
      end
    end
  end
end
```



ThoughtWorks

? ' s

please fill out the session evaluations
samples at **`github.com/nealford`**

NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
blog: memeagora.blogspot.com
twitter: neal4d

NF

Text