



Meta-Programming in Ruby

BRIAN SAM-BODDEN

@bsbodden



RUBY





Ruby Ideals...

Programming should be *fun!*



Java Developer :-)

*A programming language should
treat you like a responsible **adult***!*

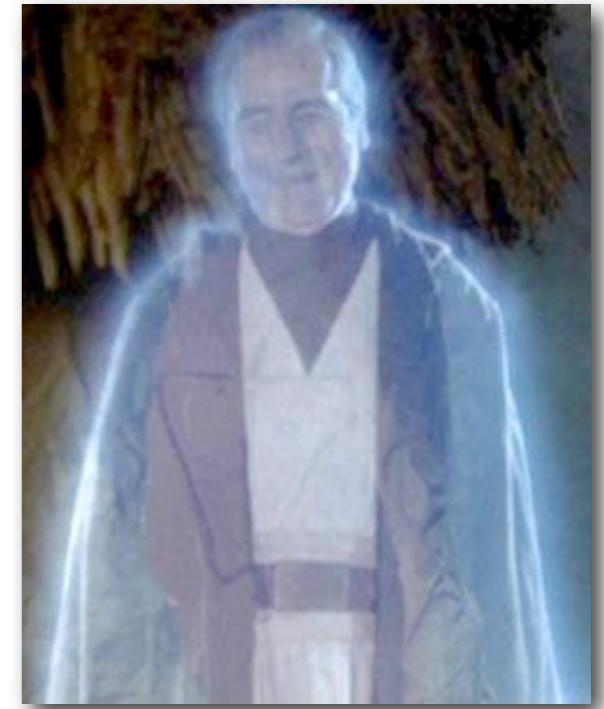
Meta-Programming



Too little



Too much



Just Enough

Without **Ruby** there would be
no **Rails**

Ruby's **meta-programming** is the **key**
feature that makes Rails magical

OPEN CLASSES



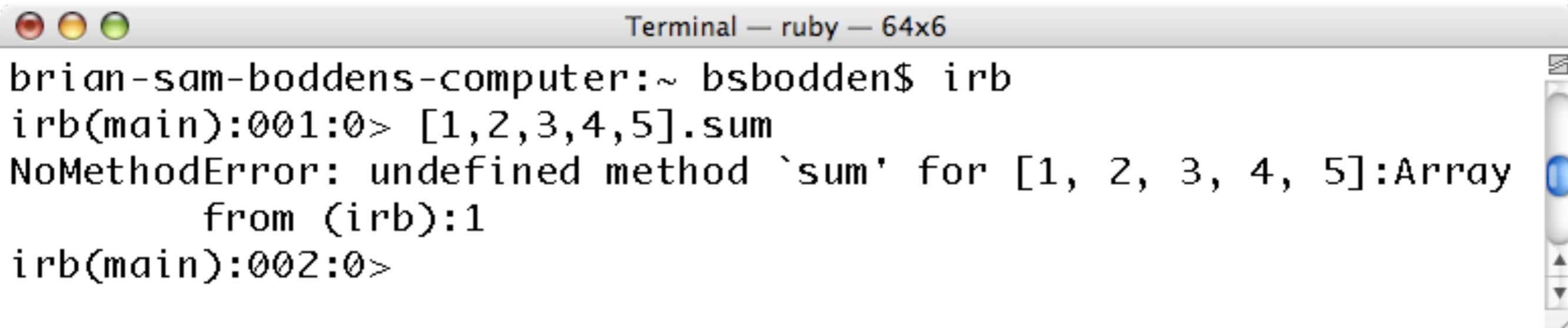
Say we have an Array like:

[1,2,3,4,5]

and we want to sum of
all of its elements

Let's try something like:

[1,2,3,4,5].sum



A screenshot of a Mac OS X terminal window titled "Terminal – ruby – 64x6". The window shows the following text:
brian-sam-boddens-computer:~ bsbodden\$ irb
irb(main):001:0> [1,2,3,4,5].sum
NoMethodError: undefined method `sum' for [1, 2, 3, 4, 5]:Array
from (irb):1
irb(main):002:0>

Rats! Doesn't work in **Ruby**

... yet!

Ruby classes are



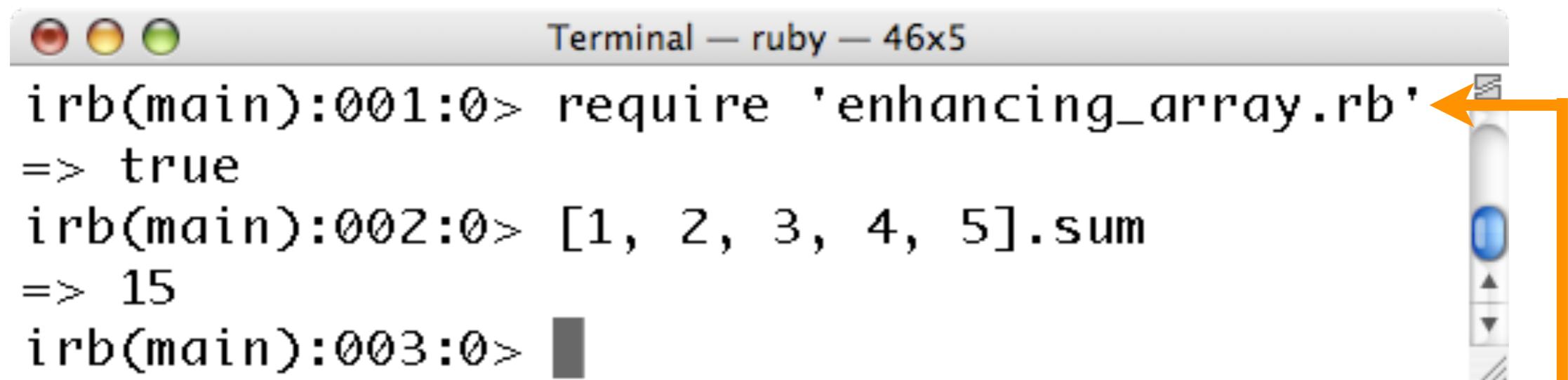
We can teach the Array class a new behavior

A screenshot of a Mac OS X-style window titled "enhancing_array.rb". The window contains a single line of Ruby code:

```
class Array
  def sum
    inject(0) { |s, v| s + v }
  end
end
```

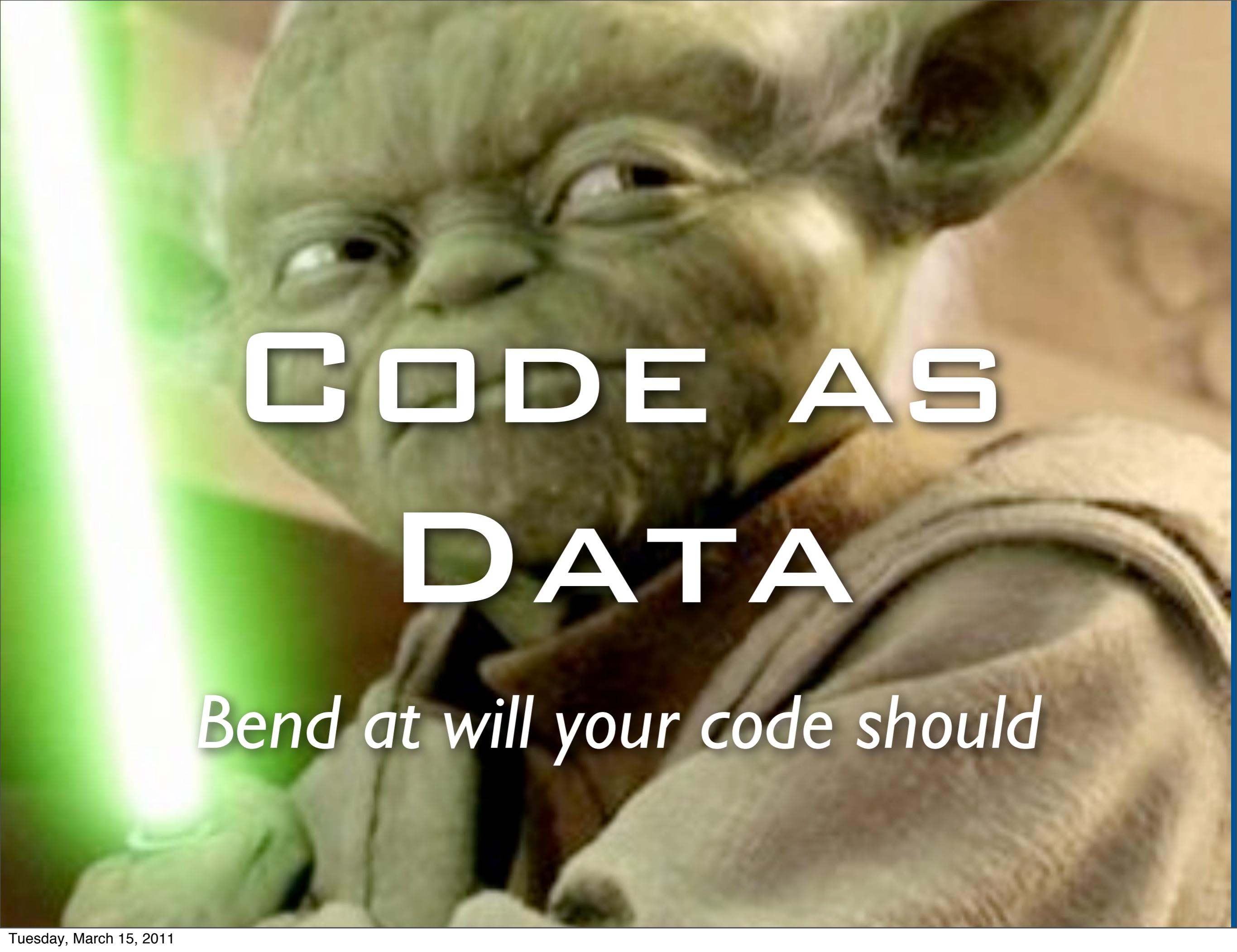
The code defines a new method "sum" for the Array class, which uses the "inject" method to calculate the sum of all elements in the array. The code editor interface includes a toolbar with icons for file operations, a sidebar with navigation buttons, and a status bar at the bottom showing "Line: 5 Column: 4" and "Ruby".

Let's try again!



```
Terminal — ruby — 46x5
irb(main):001:0> require 'enhancing_array.rb'
=> true
irb(main):002:0> [1, 2, 3, 4, 5].sum
=> 15
irb(main):003:0> ■
```

All you need to do is **load** the file containing the enhancements

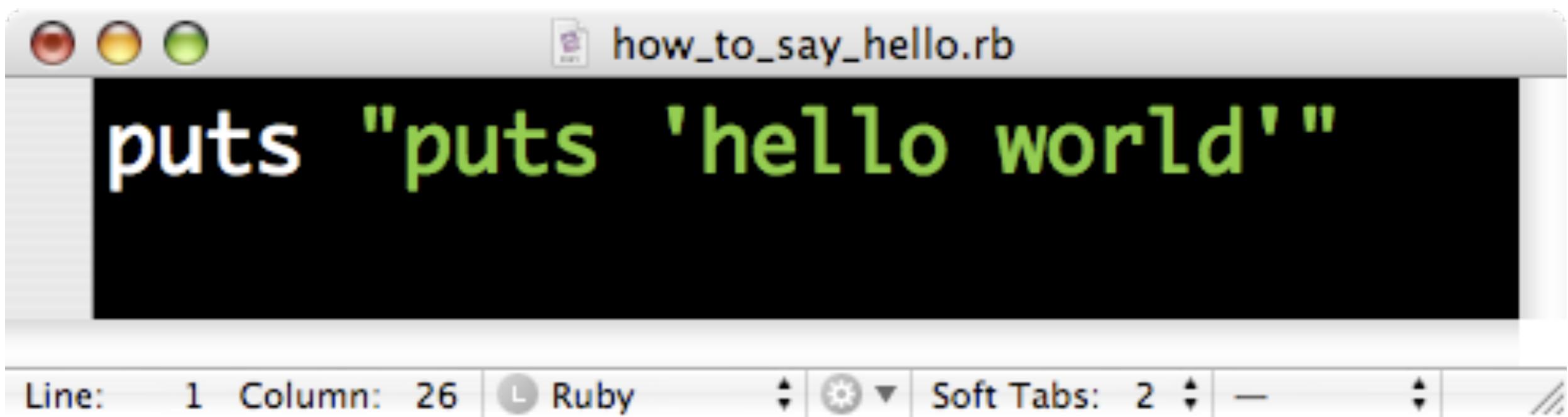


CODE AS DATA

Bend at will your code should

When **code** can be manipulated as **data**
a whole world of **possibilities** opens

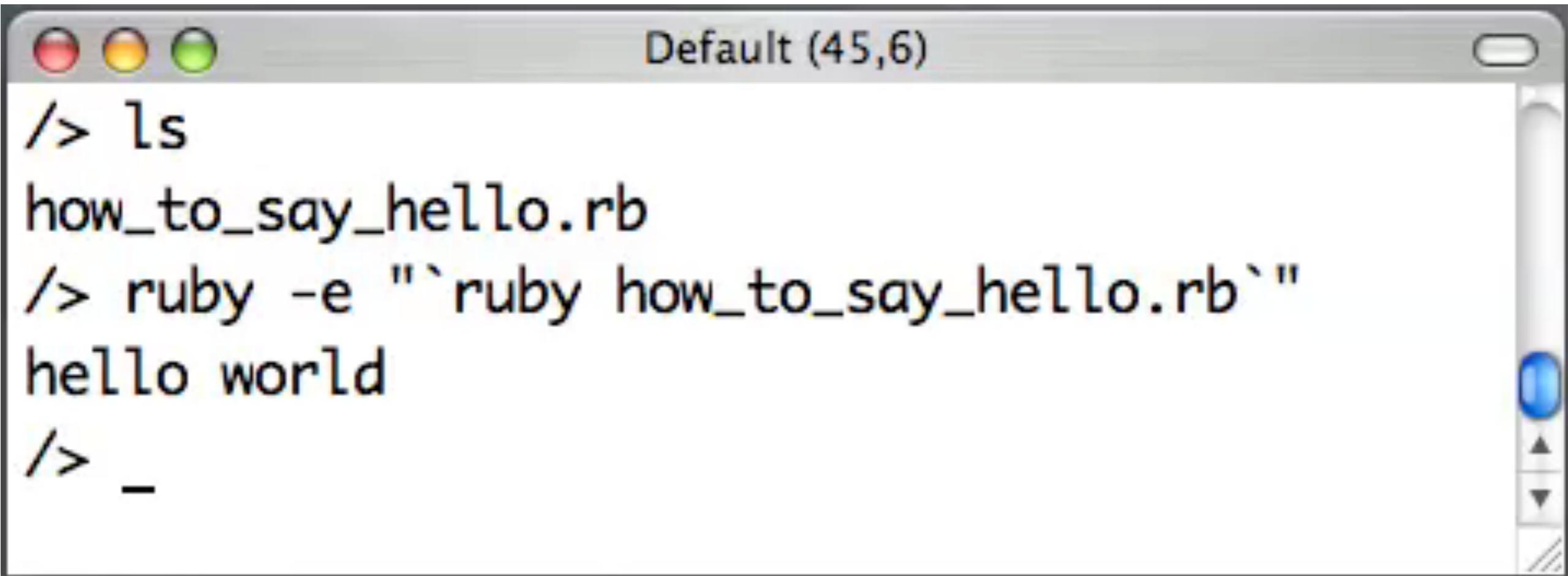
In **Ruby** you can evaluate the
contents of a string



```
how_to_say_hello.rb
puts "puts 'hello world'"
```

Line: 1 Column: 26 | L Ruby | Soft Tabs: 2 | - | //

Code that invokes code, that invokes code

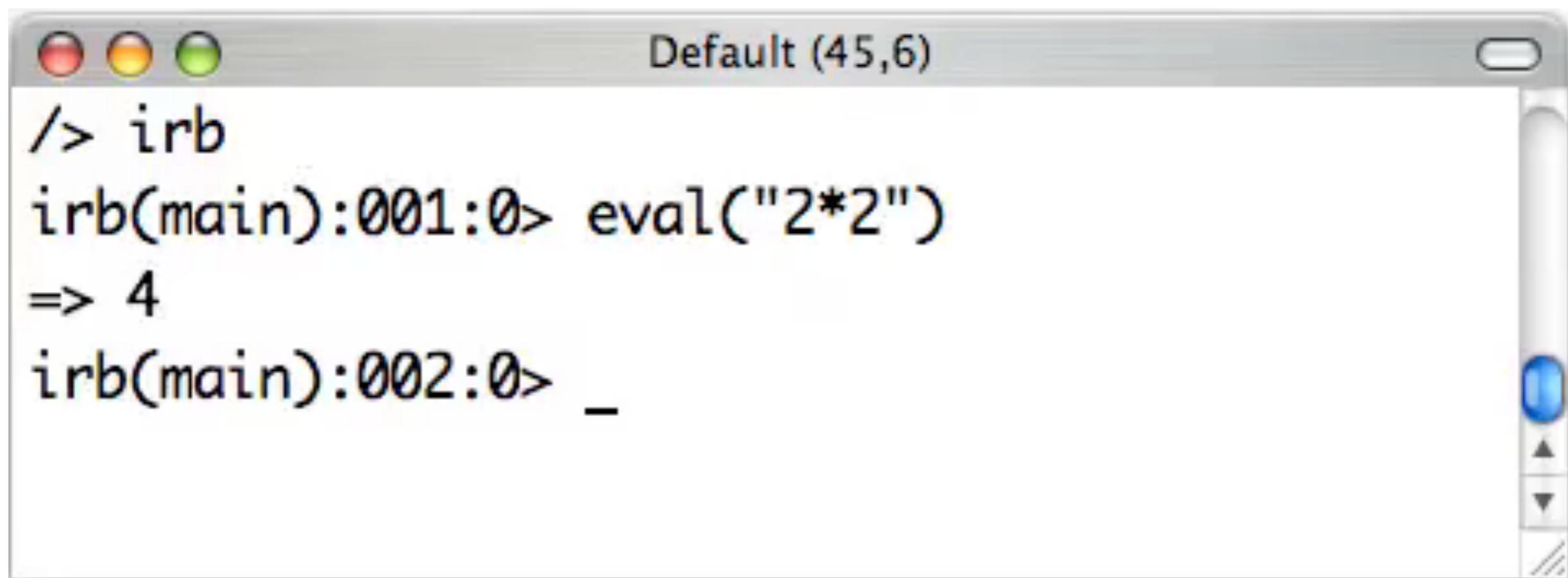


```
/> ls
how_to_say_hello.rb
/> ruby -e "`ruby how_to_say_hello.rb`"
hello world
/> _
```

The image shows a terminal window with a title bar labeled "Default (45,6)". The window contains a command-line session. The user types "/> ls" followed by the file "how_to_say_hello.rb". Then, they type "/> ruby -e "`ruby how_to_say_hello.rb`"" which runs the script and outputs "hello world". Finally, they type "/> _" which exits the terminal.

There you go, there you go

Ruby provides the **eval** family of methods for runtime execution of code stored in strings

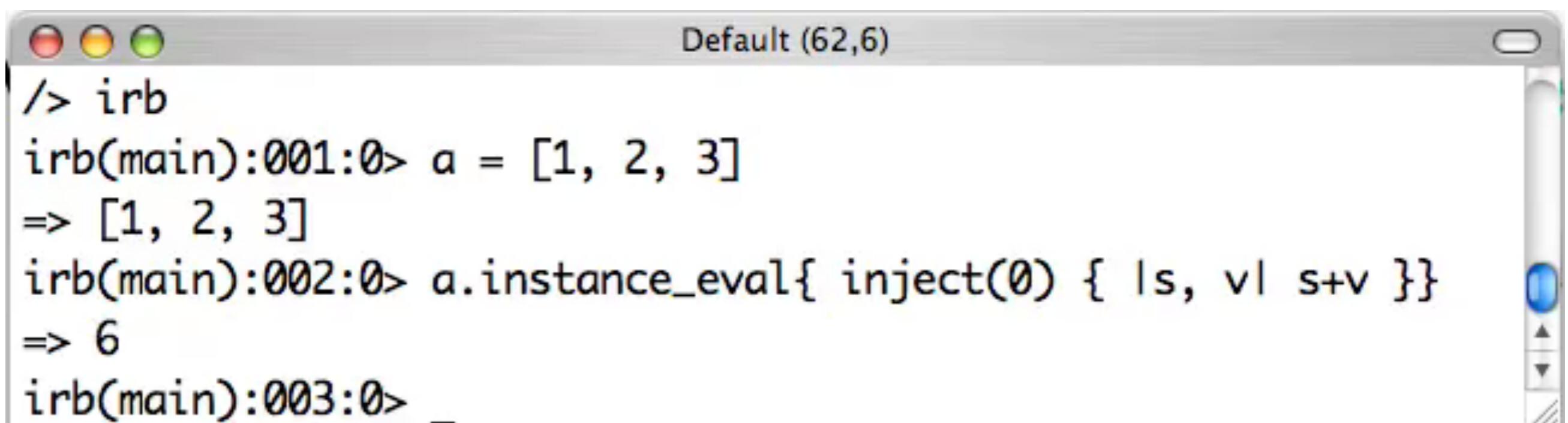


The screenshot shows a Mac OS X window titled "Default (45,6)". Inside the window, the Ruby interactive shell (irb) is running. The session starts with the command `/> irb`, followed by `irb(main):001:0> eval("2*2")`, which returns the value `=> 4`. The session ends with `irb(main):002:0> _`.

```
/> irb
irb(main):001:0> eval("2*2")
=> 4
irb(main):002:0> _
```

eval will evaluate any string

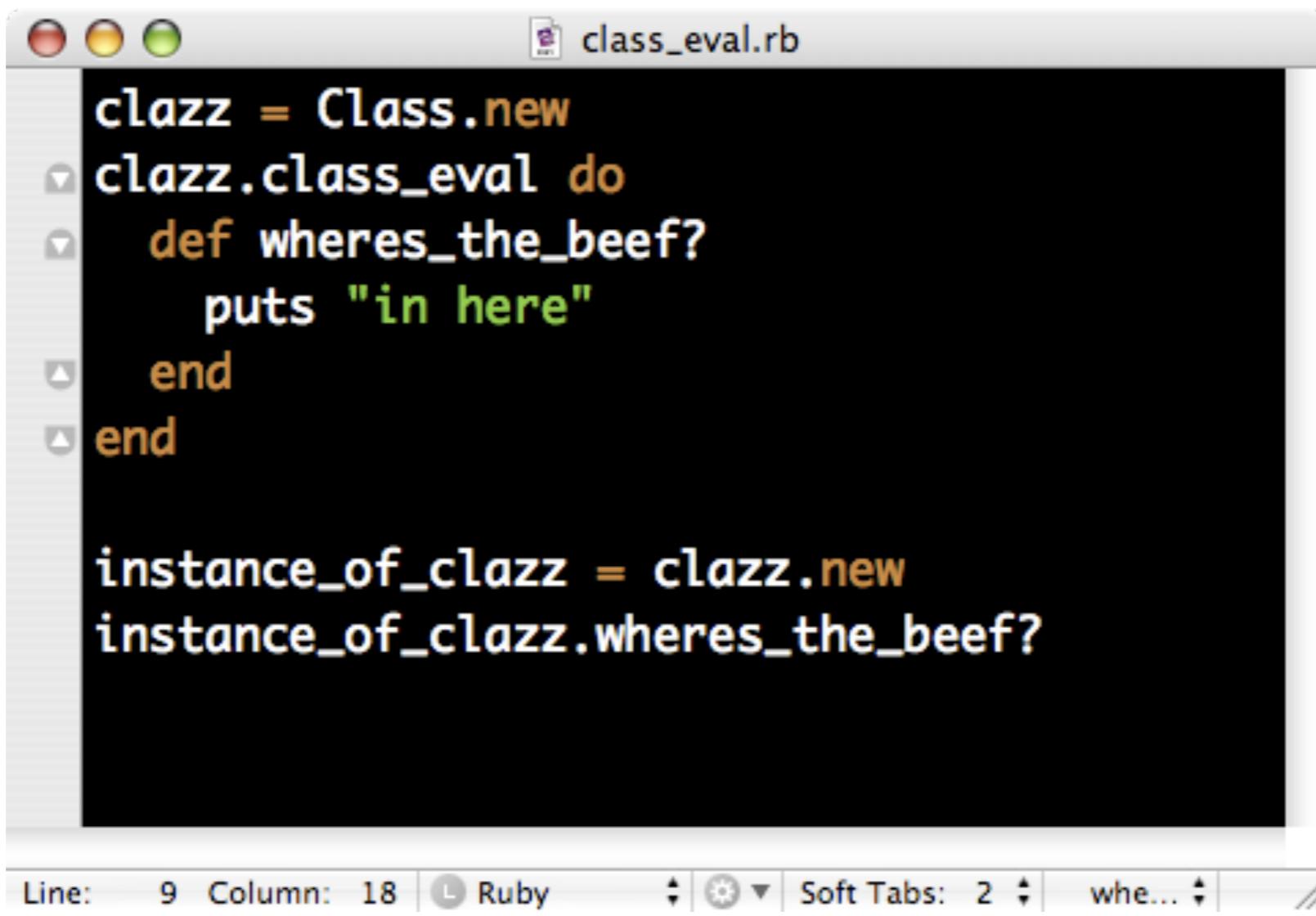
`instance_eval` can evaluate a string or a code block



```
/> irb
irb(main):001:0> a = [1, 2, 3]
=> [1, 2, 3]
irb(main):002:0> a.instance_eval{ inject(0) { |s, v| s+v } }
=> 6
irb(main):003:0>
```

The image shows a screenshot of an OS X terminal window. The window title is "Default (62,6)". The terminal is running an IRB session. The user enters "a = [1, 2, 3]" which is assigned to the variable "a". Then, the user enters "a.instance_eval{ inject(0) { |s, v| s+v } }" which evaluates to 6. The terminal has its standard OS X interface with red, yellow, and green close buttons at the top left, and scroll bars on the right.

in the context of the receiver



A screenshot of a Mac OS X-style window titled "class_eval.rb". The code editor displays the following Ruby script:

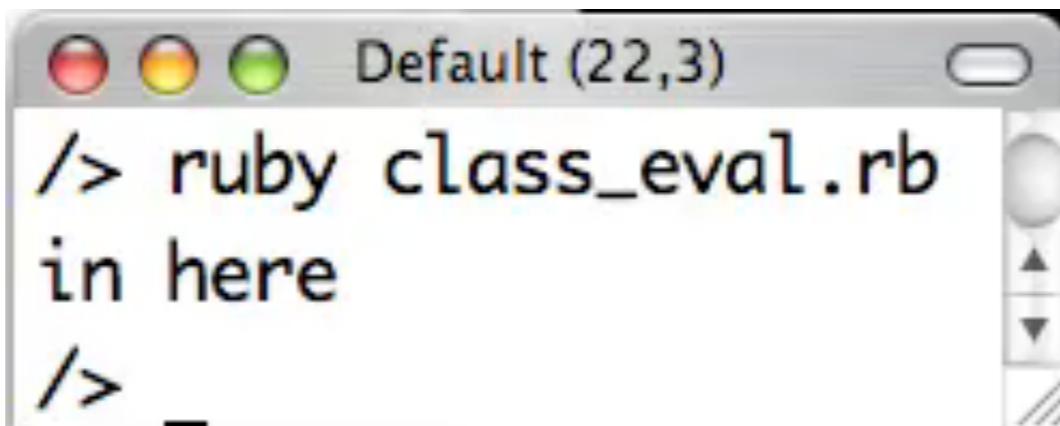
```
clazz = Class.new
clazz.class_eval do
  def wheres_the_beef?
    puts "in here"
  end
end

instance_of_clazz = clazz.new
instance_of_clazz.wheres_the_beef?
```

Line: 9 Column: 18 | L Ruby | Soft Tabs: 2 | whe... | /

class_eval can evaluate a string or a code block

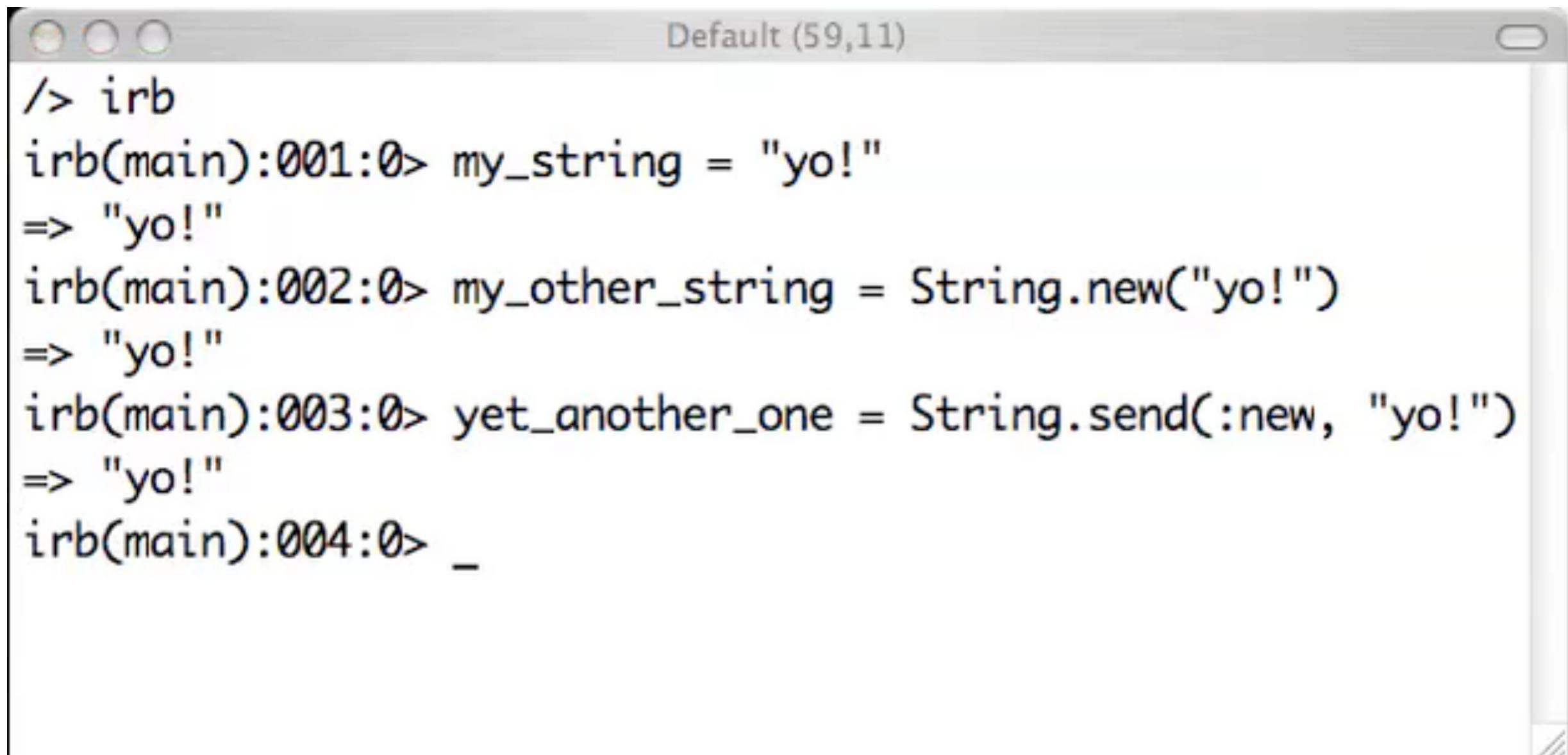
in the context of the class or module it is called on



A screenshot of a terminal window titled "Default (22,3)". The terminal shows the command "/> ruby class_eval.rb" followed by the output "in here" and then the prompt "/>".

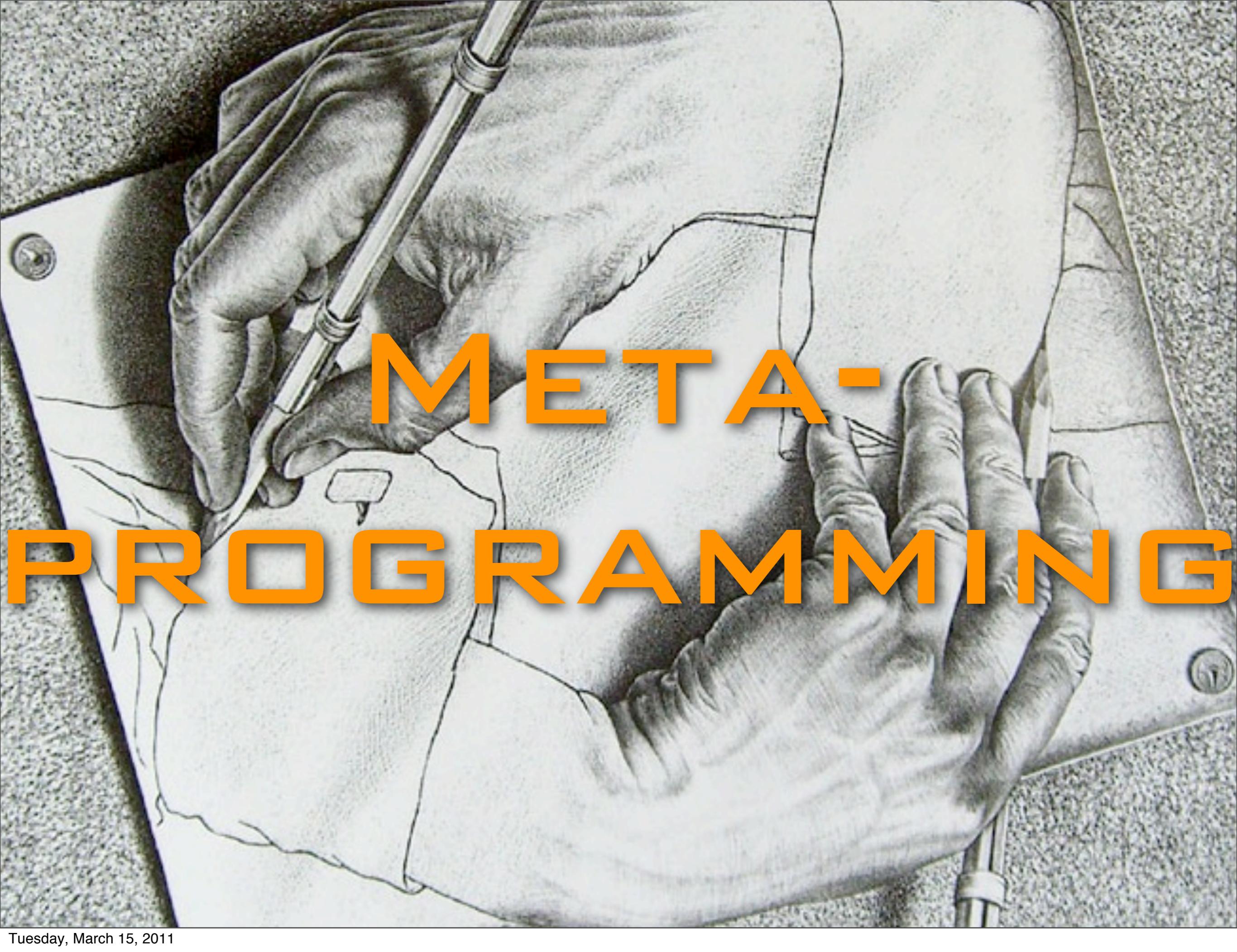
```
/> ruby class_eval.rb
in here
/>
```

In **Ruby** we try to think about sending a message to an object rather than calling a method on an object



```
/> irb
irb(main):001:0> my_string = "yo!"
=> "yo!"
irb(main):002:0> my_other_string = String.new("yo!")
=> "yo!"
irb(main):003:0> yet_another_one = String.send(:new, "yo!")
=> "yo!"
irb(main):004:0> _
```

In simple cases (above) syntactic sugar hides it, but we can use it in interesting ways...



META- PROGRAMMING

Meta-programming

...is about programs that
write programs

...it's a superb tool for building
frameworks

...it's the key ingredient for building
domain-specific languages

Ruby's is a great vehicle for meta-programming because it is:

dynamic and reflexive

open and malleable

code is data, data is code

clean* syntax

programming event model



Ruby on Rails

...uses meta-programming to bring
the language closer to the problem

...is a collection of domain-specific
languages (DSL) for building web
applications

SINGLETON CLASS

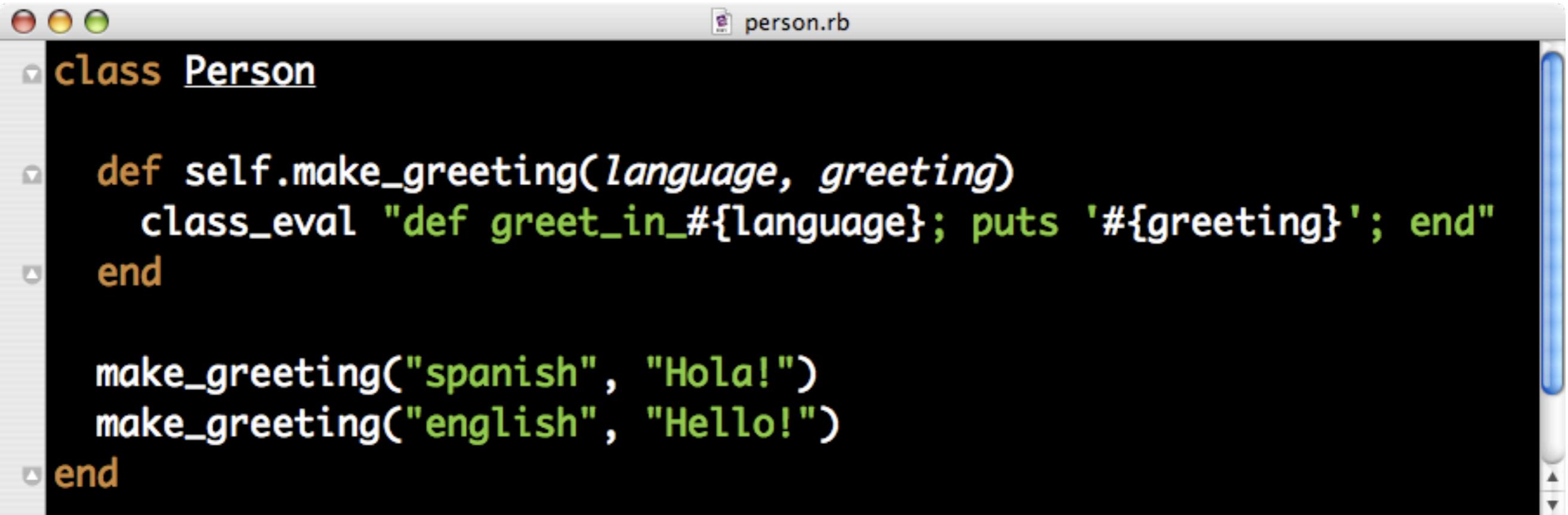


In **Ruby** you can enhance a particular **instance** of a class

Ruby uses a **proxy** class known as the **singleton** class

Meta-class: The **singleton** for **Class** objects

class_eval indirectly uses the meta-class



```
class Person

  def self.make_greeting(language, greeting)
    class_eval "def greet_in_#{language}; puts '#{greeting}'; end"
  end

  make_greeting("spanish", "Hola!")
  make_greeting("english", "Hello!")
end
```

Line: 11 Column: 1 Ruby Soft Tabs: 2 self.make_greeting(language, greeting)

adding a singleton method to the class

Rails relies heavily on **Ruby's** ability to dynamically enhance a class

```
Default (79,16)
irb(main):003:0> bob.greet_in_spanish
Hola!
=> nil
irb(main):004:0> bob.greet_in_english
Hello!
=> nil
irb(main):005:0> clazz = Person
=> Person
irb(main):006:0> clazz.make_greeting('dutch', 'Hallo!')
=> nil
irb(main):007:0> jim = clazz.new
=> #<Person:0x57214>
irb(main):008:0> jim.greet_in_dutch
Hallo!
=> nil
irb(main):009:0>
```

Shortcuts such as *define_method* exist to make life easier



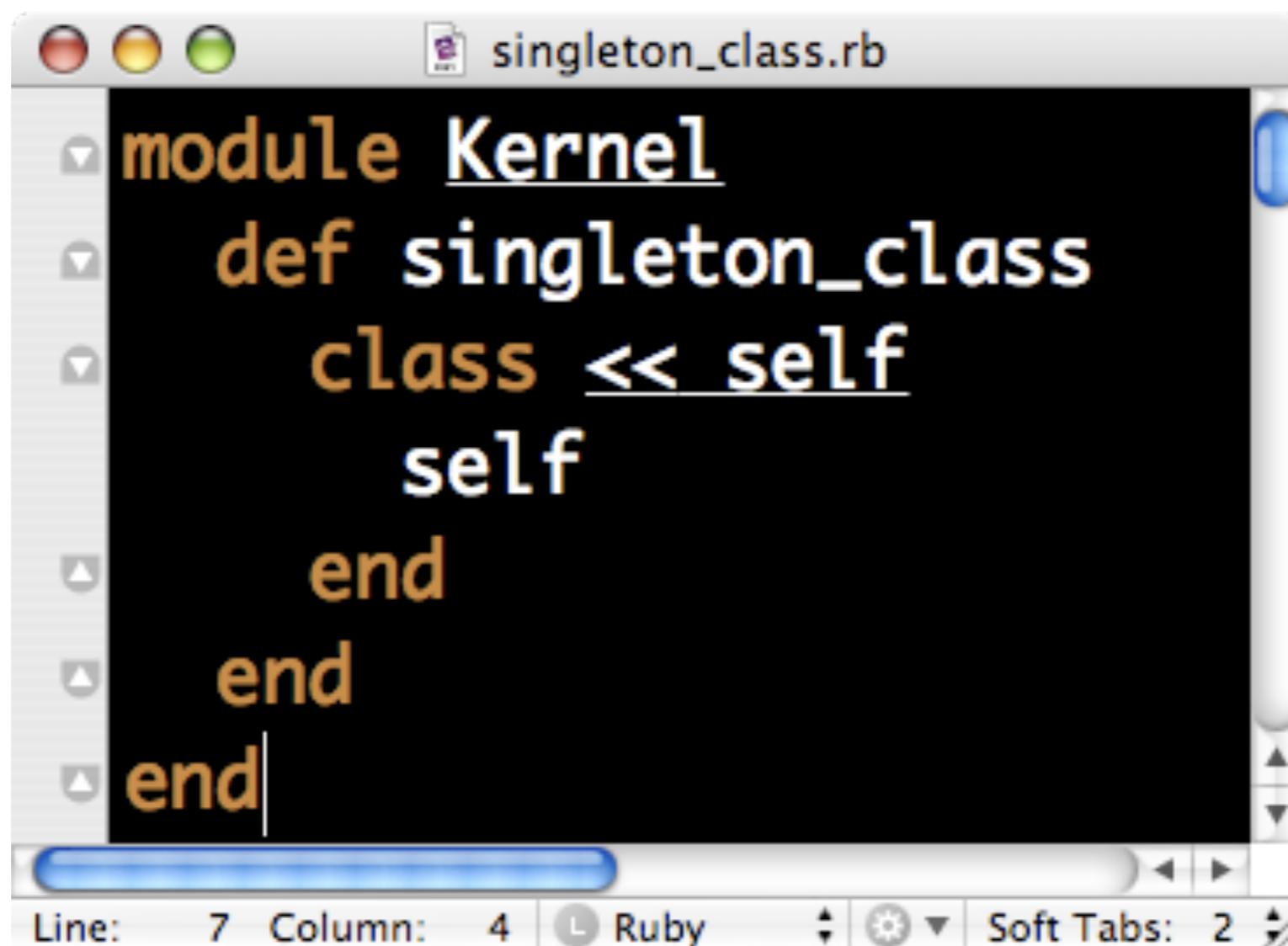
```
class Person

  def self.make_greeting(language, greeting)
    define_method "greet_in_#{language}" do
      puts "#{greeting}"
    end
  end

  make_greeting("spanish", "Hola!")
  make_greeting("english", "Hello!")
end
```

Line: 11 Column: 4 | L Ruby | Soft Tabs: 2 | self.make_greeting(language, greeting)

Accessing the singleton meta-class explicitly



A screenshot of a Mac OS X-style window titled "singleton_class.rb". The window contains a single line of Ruby code:

```
module Kernel
  def singleton_class
    class << self
      self
    end
  end
end
```

The code defines a module named "Kernel" and within it, a method named "singleton_class". This method creates a new class that inherits from the current object ("self") and returns it. The entire code block is enclosed in "end" statements.

Line: 7 Column: 4 | L Ruby | Soft Tabs: 2

Child classes get enhanced with class methods

```
class Application
```

```
# specific applications get a singleton class level instances for holding
# the array of pages contained in the application and a the home page
def self.inherited(child) #:nodoc:
  metaclass = Utils.metaclass(child)
  Utils.attr_array(child, :pages, :create_accessor => false)
  metaclass.instance_eval do
    define_method(:homepage) { @homepage }
    define_method(:homepage=) { |page| @homepage = page }
  end
  super
end
```

```
module Utils
```

```
def self.metaclass(clazz = self)
  class << clazz; self; end
end
```



INHERITANCE THE RUBY WAY

Many Ruby DSLs create class methods on subclasses using the meta-class (singleton class)

Rails uses this technique in ActiveRecord

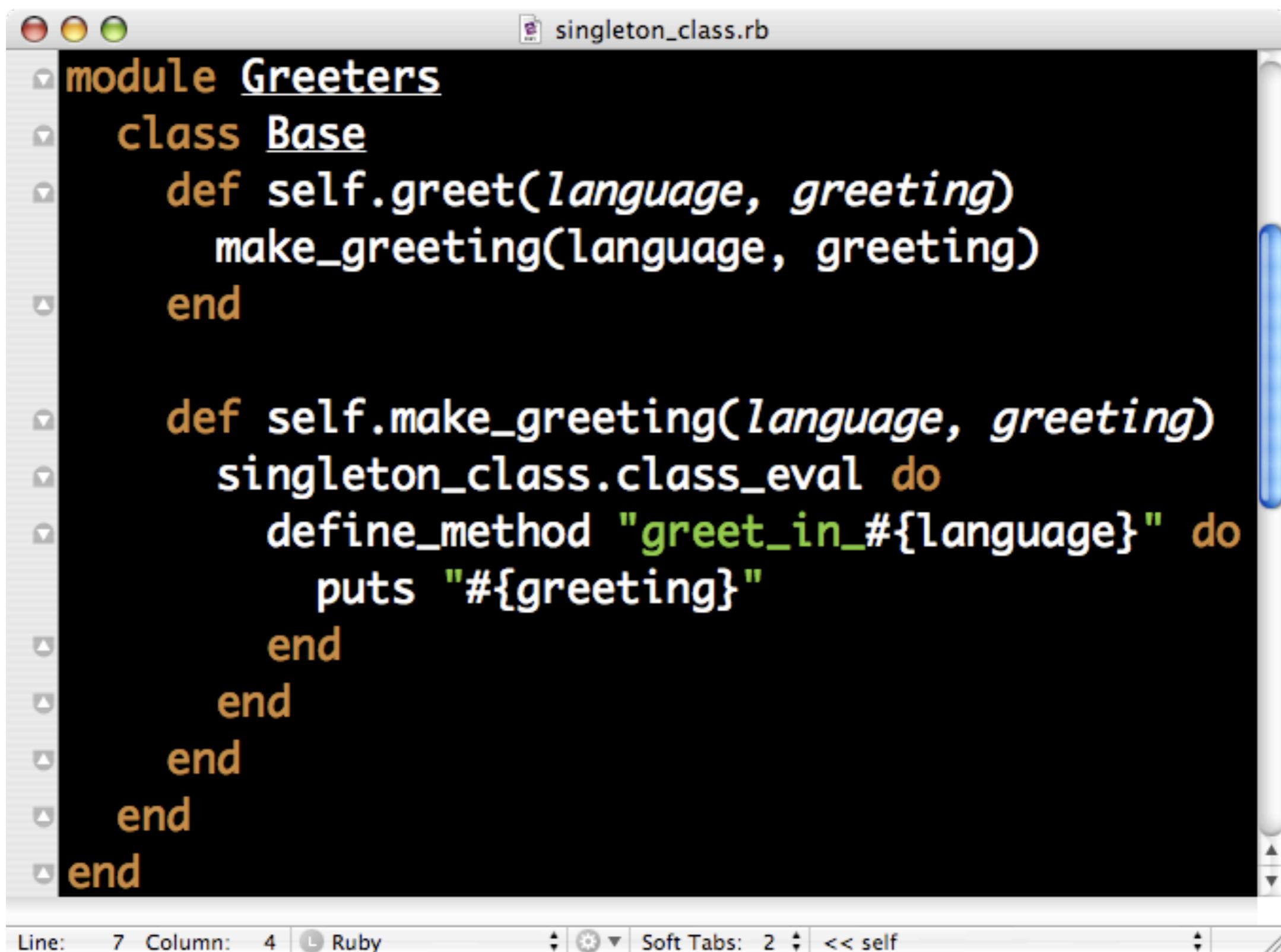
```
class ProjectsUser < ActiveRecord::Base
  belongs_to :project
  belongs_to :user

  ROLES = { 1 => "Worker", 2 => "Manager" }

  validates_uniqueness_of :project_id, :scope => :user_id
  validates_uniqueness_of :user_id, :scope => :project_id

  def to_s
    "#{user.login} (#{ROLES[role_type]})"
  end
end
```

A module with a “Base” class

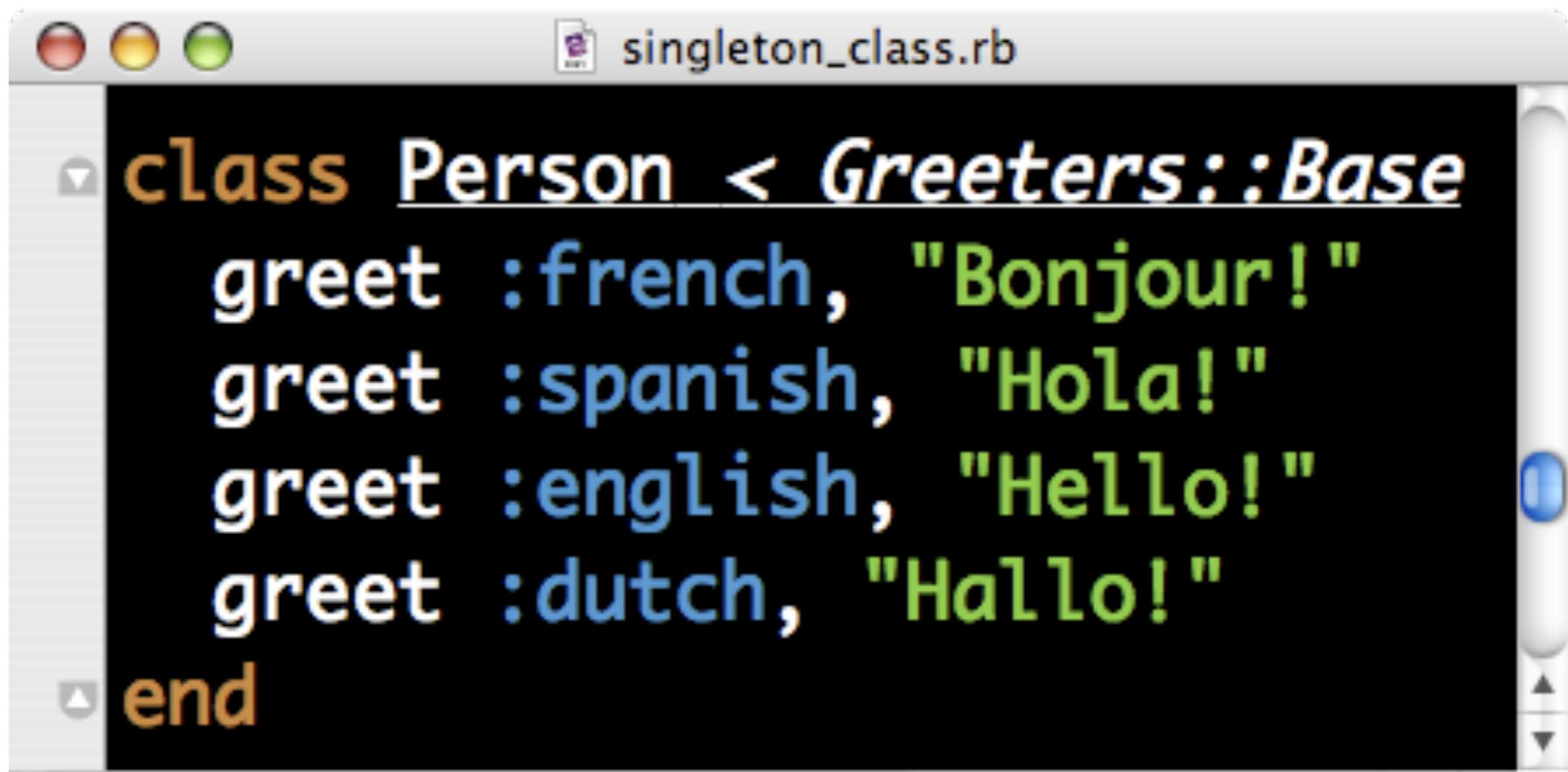


```
module Greeters
  class Base
    def self.greet(language, greeting)
      make_greeting(language, greeting)
    end

    def self.make_greeting(language, greeting)
      singleton_class.class_eval do
        define_method "greet_in_#{language}" do
          puts "#{greeting}"
        end
      end
    end
  end
end
```

Line: 7 Column: 4 | Ruby | Soft Tabs: 2 | << self

Methods get added to our classes with simple declarations



A screenshot of a Mac OS X TextEdit window titled "singleton_class.rb". The window contains the following Ruby code:

```
class Person < Greeters::Base
  greet :french, "Bonjour!"
  greet :spanish, "Hola!"
  greet :english, "Hello!"
  greet :dutch, "Hallo!"
end
```

The code defines a class `Person` that inherits from `Greeters::Base`. It contains four methods named `greet` with parameters `:french`, `:spanish`, `:english`, and `:dutch`, each returning a string greeting in that language.

The TextEdit window has a dark theme. The status bar at the bottom shows "Line: 7 Column: 4" and "Ruby".

Getting closer to a DSL!

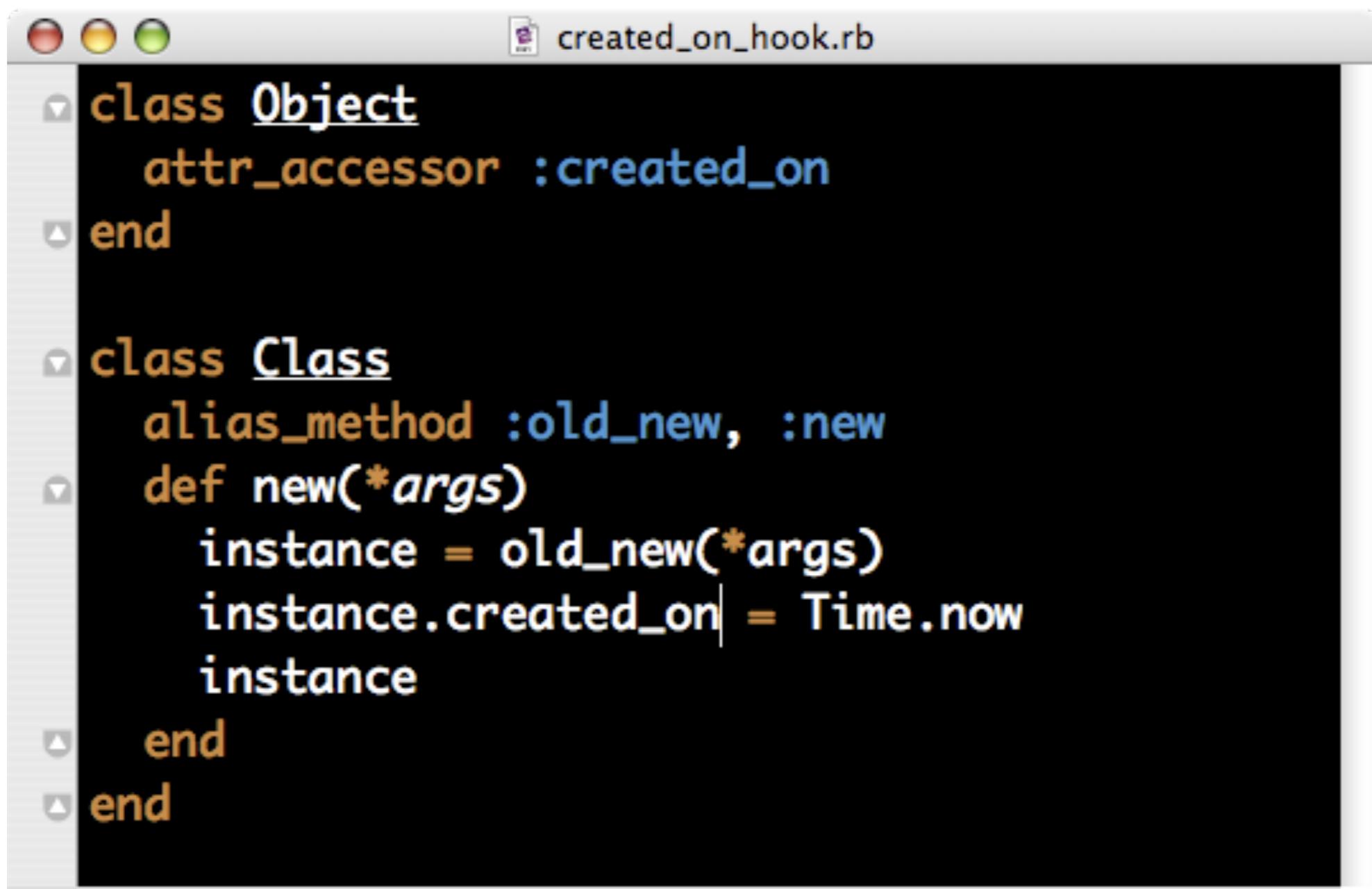
Now our class has a new set of class methods

```
Default (79,16)
/> irb
irb(main):001:0> require 'singleton_class.rb'
=> true
irb(main):002:0> Person.greet_in_french
Bonjour!
=> nil
irb(main):003:0> Person.greet_in_spanish
Hola!
=> nil
irb(main):004:0> Person.greet_in_english
Hello!
=> nil
irb(main):005:0> Person.greet_in_dutch
Hallo!
=> nil
irb(main):006:0> _
```



AOP
THE RUBY
WAY

With *alias_method* you can wrap an existing method...



The screenshot shows a code editor window with a dark theme. The title bar says "created_on_hook.rb". The code in the editor is:

```
class Object
  attr_accessor :created_on
end

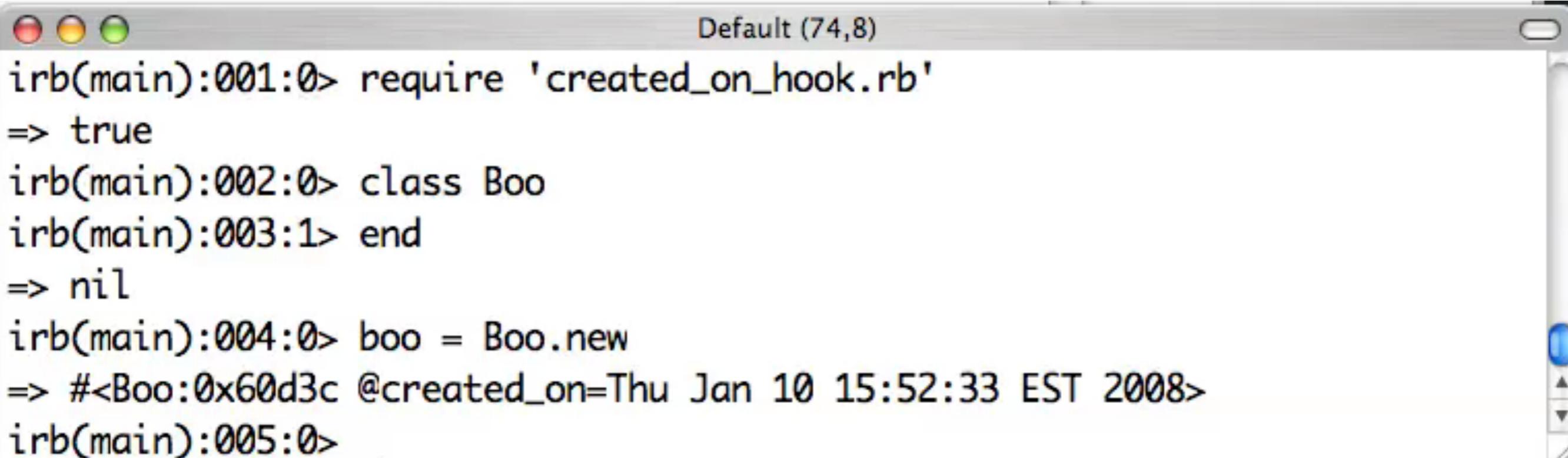
class Class
  alias_method :old_new, :new
  def new(*args)
    instance = old_new(*args)
    instance.created_on = Time.now
    instance
  end
end
```

The code defines two classes: `Object` and `Class`. The `Object` class has an `attr_accessor` for `created_on`. The `Class` class wraps the `:new` method with `:old_new`, creating a new instance and setting its `created_on` attribute to the current time before returning it.

At the bottom of the editor, there is a status bar with the following information:

Line: 9 Column: 24 | Ruby | Soft Tabs: 2 | new(*args)

...effectively intercepting any calls
and injecting any desired behavior



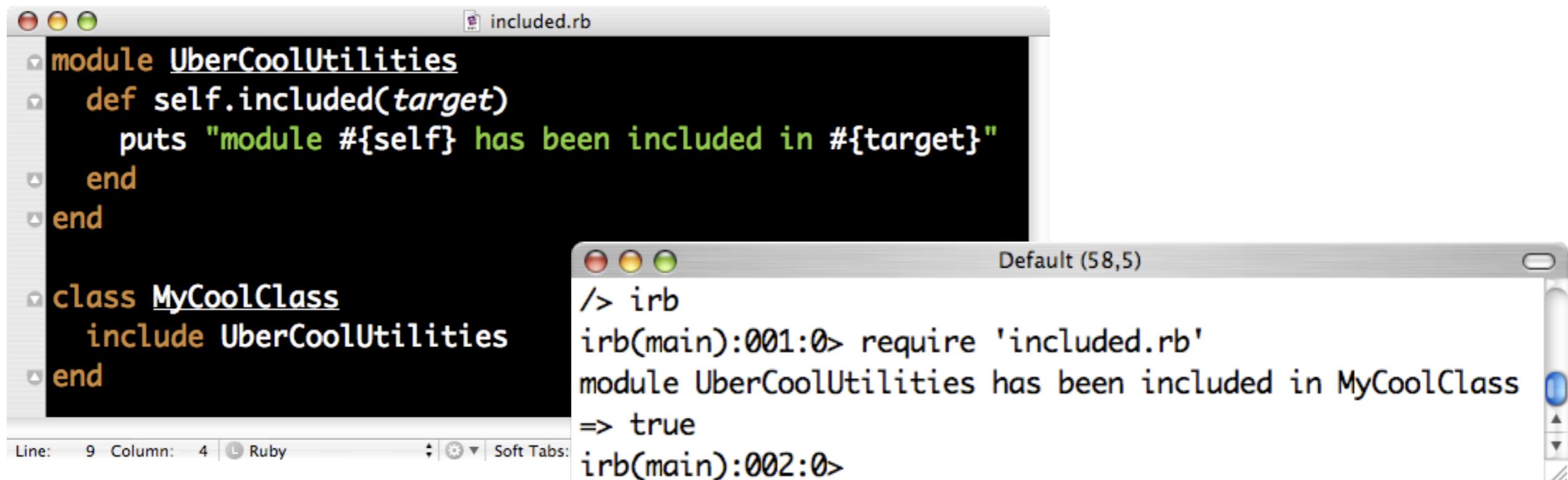
The screenshot shows a Mac OS X terminal window titled "Default (74,8)". The window contains the following IRB session:

```
irb(main):001:0> require 'created_on_hook.rb'
=> true
irb(main):002:0> class Boo
irb(main):003:1> end
=> nil
irb(main):004:0> boo = Boo.new
=> #<Boo:0x60d3c @created_on=Thu Jan 10 15:52:33 EST 2008>
irb(main):005:0> _
```

META-PROGRAMMING EVENTS

INCLUDED HOOK

Ruby has a rich event model associated with static and dynamic changes of the code



The image shows a Mac OS X desktop environment. In the top-left corner, there is a code editor window titled "included.rb". The code inside is:

```
module UberCoolUtilities
  def self.included(target)
    puts "module #{self} has been included in #{target}"
  end
end

class MyCoolClass
  include UberCoolUtilities
end
```

In the bottom-right corner, there is an IRB (Interactive Ruby) window. The session starts with "/> irb", followed by "irb(main):001:0> require 'included.rb'", then "module UberCoolUtilities has been included in MyCoolClass", then "=> true", and finally "irb(main):002:0>".



METHOD MISSING

Ruby provides several hook methods that can be used to create custom behaviors

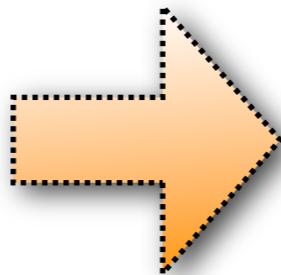
method_missing

method_added

method_removed

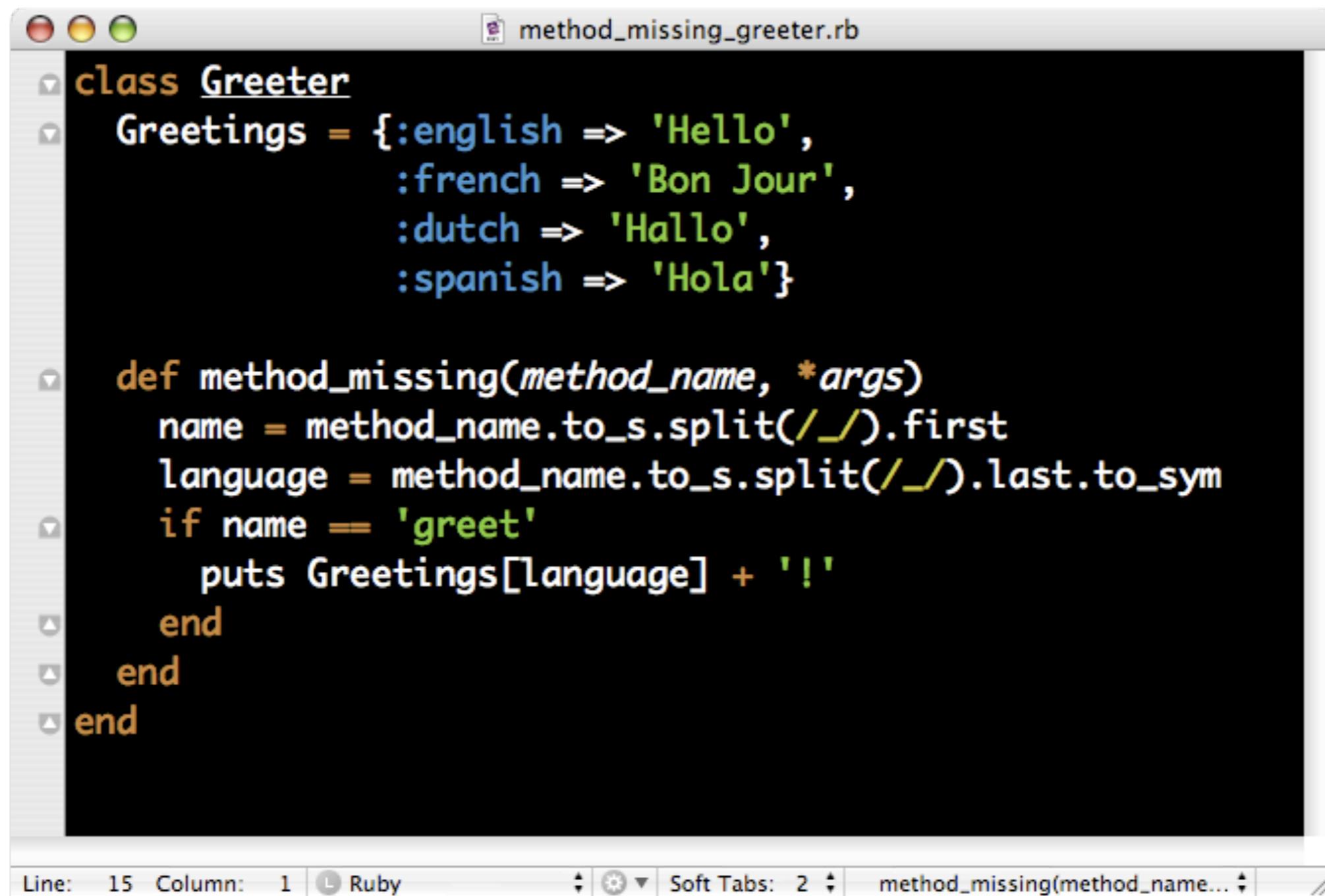
Ruby's Markup Builder is an example of what can be achieved with method missing

```
require 'rubygems'  
require 'builder'  
  
builder = Builder::XmlMarkup.new  
  
builder.html {  
  builder.head {  
    builder.title "History"  
  }  
  builder.body {  
    builder.h1 "Header"  
    builder.comment! "A Comment"  
  }  
}
```



```
<html>  
  <head>  
    <title>  
      History  
    </title>  
  </head>  
  <body>  
    <h1>  
      Header  
    </h1>  
    <!-- A Comment -->  
  </body>  
</html>
```

Let's implement the greeter example using *method_missing*



The screenshot shows a code editor window titled "method_missing_greeter.rb". The code defines a class named "Greeter" with a hash variable "Greetings" mapping languages to greetings. It includes a "method_missing" method that extracts the language from the method name and prints the corresponding greeting. The code is as follows:

```
class Greeter
  Greetings = { :english => 'Hello',
    :french => 'Bon Jour',
    :dutch => 'Hallo',
    :spanish => 'Hola' }

  def method_missing(method_name, *args)
    name = method_name.to_s.split(/_/.first
language = method_name.to_s.split(/_/.last.to_sym
if name == 'greet'
  puts Greetings[language] + '!'
end
end
end
```

The status bar at the bottom indicates "Line: 15 Column: 1" and "Ruby".

greeter using *method_missing*

```
Default (75,15)
/> irb
irb(main):001:0> require 'method_missing_greeter.rb'
=> true
irb(main):002:0> greeter = Greeter.new
=> #<Greeter:0x62cb8>
irb(main):003:0> greeter.greet_in_dutch
Hallo!
=> nil
irb(main):004:0> greeter.greet_in_french
Bon Jour!
=> nil
irb(main):005:0> _
```



I was bored...

In the Ruby world I've worked with Rails and Merb

...thinking of web apps in terms of request & response

meanwhile... in the Java world I was working with
Tapestry and Wicket

...thinking about Pages, Components and Events



Started to hack to see what it would be to build something like Tapestry/Wicket in **Ruby**

...actually it all started as an learning exercise

...part of a conversation among friends about the need for IoC and AOP frameworks for dynamic languages

...but then I kept going

...somehow I ended building component-oriented web framework in Ruby

**YOU MIGHT BE
ASKING YOURSELF,
BUT WHY?**

WHY NOT?

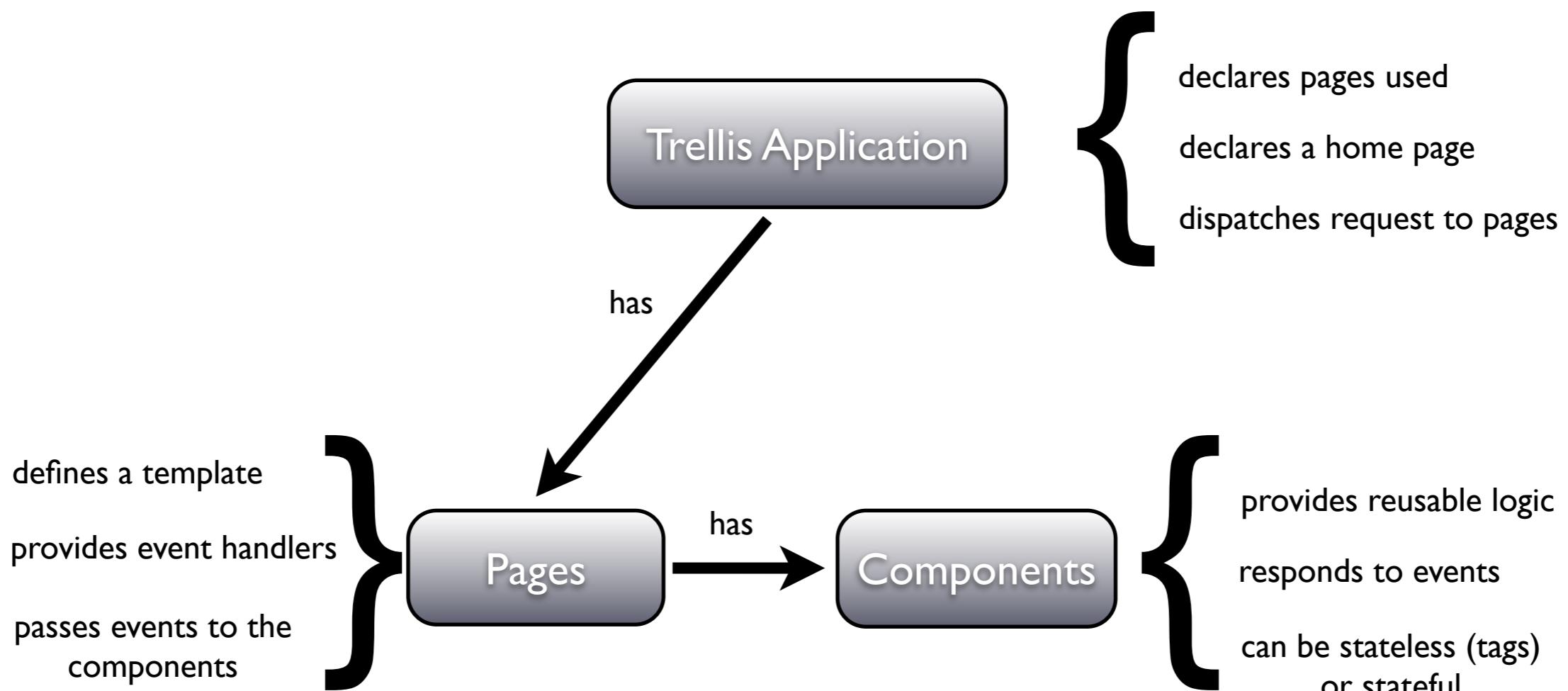




My goals:

- ✓ Magic like Rails; less code more action
- ✓ Small like Camping... a micro-framework
- ✓ Components like Tapestry/Wicket
- ✓ Clean HTML templates...
- ✓ ...but also programmatic HTML generation
- ✓ Non-managed components, no pooling

The big picture:





The simplest Trellis App, one Application and one Page

```
require 'trellis/trellis'

include Trellis

module Simplest

  class Simplest < Application
    home :start
  end

  class Start < Page
    attr_accessor :current_time

    def initialize
      @current_time = Time.now
    end
  end

  Simplest.new.start
end
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Simplest Trellis Application</title>
  </head>
  <body>
    <h1>Simplest Start Page</h1>

    <p>This is the start page for this application, a good place to start your modifications.  

       Just to prove this is live: </p>

    <p>The current time is: <trellis:value name="current_time">${currentTime}</trellis:value> </p>

    <p>[<trellis:page_link tpage="start">refresh</trellis:page_link>]</p>
  </body>
</html>
```

The Template

The Code



What did I use to put this thing together...

- ➡ A boat load of meta-programming
- ➡ Rack ...HTTP the Ruby way
- ➡ Radius ...XML tags for the templates
- ➡ Builder ...for building HTML/XML
- ➡ Hpricot, REXML ...parsing searching HTML/XML
- ➡ Paginator ...for duh, pagination
- ➡ Parts of Rails ...so far ActiveSupport (Inflectors)



Let's do some code spelunking on the **Trellis** codebase...

yes, there are some things that I'm not proud of but...

...this is pre-alpha, unreleased v0.000001;-)



Some of the example applications that I've put together in order of complexity

→ hilo

→ stateful_counters

→ guest_book

→ flickr

→ hangman

→ simple_blog

→ yui

→ crud_components



WHAT HAVE
WE LEARNED?

There is more to **Ruby** than Rails!

Building a Framework?
Give **Ruby** a try!

Build the language up towards
the problem