Advanced Ruby Class Design

Jim Weirich
Chief Scientist
EdgeCase
software artisans

Advanced?

Ruby!

Where I Come From

FORTRAN
C
Modula 2
C++
Eiffel
Java

Ruby

Lisp
FORTH
TCL
Perl

A Real Programmer can write Java code in *any* language!

Thinking

Thinking

Ruby Class Design:
What to Expect

Three Examples of (more or less) Real Life Ruby Classes

Interesting and/or Fun (at least to me)

Illustrate Techniques that are not typically used by the Java/C++/Eiffel Crowd

Box 1
Master of Disguise

Rake::FileList

```
RUBY_FILES = FileList[‘lib/**/*.rb’]
```

FileList is like an Array, except:

- Initialized with GLOB
- Specialized to_s
- Extra Methods (ext, pathmap, etc)
- Lazy Evaluation

First Cut

```
class FileList < Array
  ...
end
```

Lazy Loading

```
def initialize(pattern)
  super
  @pattern = pattern
  @resolved = false
end
```

Lazy Loading

```
def resolve
  self.clear
  Dir[@pattern].each do |arg|
    self << arg
  end
  @resolved = true
end
```

This Will Not Work!

```
fl = FileList.new("*.c")

assert_equal 'c.c', fl[0]
```

Need to Resolve!

```
fl = FileList.new("*.c")
fl.resolve
assert_equal 'c.c', fl[0]
```

Major Pain

Auto Resolve

```
  def [](index)
    resolve unless @resolved
    super
  end
```

Yuck ... A lot of methods need resolving

Wash, Rinse, Repeat ...

```
def [](index)  ... end
def size       ... end
def empty?     ... end
def +(other)   ... end
```

A lot of methods need AutoResolve!

So, Everything is Good.

Right?

This is OK

No problem, FileList#+ is a resolving method

```
fl = FileList.new("*.rb") # picks up a.rb

new_list = fl + ["main.rb"]

new_list ==> ["a.rb", "main.rb"]
```

But this is a Small Problem

Oops ... Array#+ does not resolve its arguments

```
fl = FileList.new("*.rb") # picks up a.rb

new_list = ["main.rb"] + fl

new_list ==> ["main.rb"]
```

So the new list has the WRONG result

Why?

Because

- The Ruby implementation of Array#+ thinks its argument is an Array.
  - After all, it is (it is a subclass of Array)
- So the Array contents are used directly, rather than being resolved.

If only ...

... there was a way for an arbitrary object to indicate that it wished to be treated as an Array.

to_ary

Change this ...

```
class FileList < Array
  def initialize(pattern=nil)
    super
    @pattern = pattern
    @resolved = false
  end
  ...
```

... to this

```
class FileList
  def initialize(pattern=nil)
    @items = []
    @pattern = pattern
    @resolved = false
  end
  ...
```

Change resolving from this ...

```
  def [](index)
    resolve unless @resolved
    super
  end
```

... to this

```
def [](index)
  resolve unless @resolved
  @items[index]
end
```

But this is a Small Problem

Now ... Everything is Good

```
fl = FileList.new(“*.rb”) # picks up a.rb

new_list = [“main.rb”] + fl

new_list ==> [“main.rb”, “a.rb”]
```

Remember?

```
def [](index)  ... end
def size       ... end
def empty?     ... end
def +(other)   ... end
```

A lot of methods need AutoResolve!

Time to DRY

... to this

```
RESOLVING_METHODS =
   [:[], :size, :empty?, +:, ...]
```

```
  RESOLVING_METHODS.each do |method|
    class_eval %{
      def #{method}(*args, &block)
        resolve unless @resolved
        @items.#{method)(*args, &block)
      end
    }
  end
```

What have we learned?

When trying to mimic a class ...

it might be better to use
to_ary / to_str
rather than inheritance.

Box 2
The Art of Doing Nothing

Builder::XmlMarkup

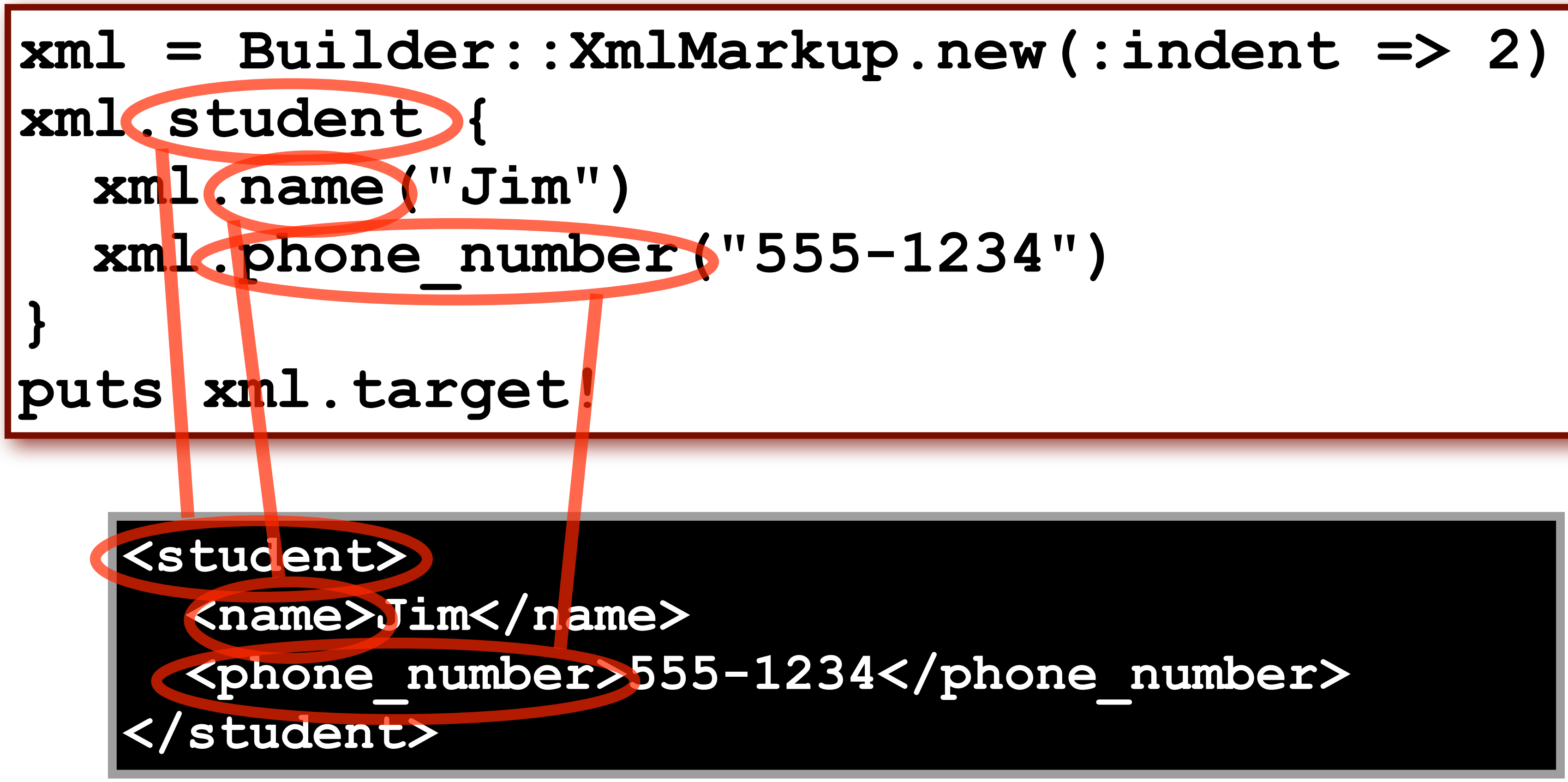
```
xml = Builder::XmlMarkup.new(:indent => 2)
xml.student {
  xml.name("Jim")
  xml.phone_number("555-1234")
}
puts xml.target!
```

Builder::XmlMarkup

```ruby
xml = Builder::XmlMarkup.new(:indent => 2)
xml.student {
  xml.name("Jim")
  xml.phone_number("555-1234")
}
puts xml.target!
```

```
<student>
  <name>Jim</name>
  <phone_number>555-1234</phone_number>
</student>
```

Builder::XmlMarkup



```ruby
xml = Builder::XmlMarkup.new(:indent => 2)
xml.student {
  xml.name("Jim")
  xml.phone_number("555-1234")
}
puts xml.target!
```

```
<student>
  <name>Jim</name>
  <phone_number>555-1234</phone_number>
</student>
```

Depends on method_missing to construct tags.

Builder::XmlMarkup

```
xml = Builder::XmlMarkup.new(:indent => 2)
xml.student {
  xml.name("Jim")
  xml.phone_number("555-1234")
  xml.class("Intro to Ruby")
}
puts xml.target!
```

Builder::XmlMarkup

```
xml = Builder::XmlMarkup.new(:indent => 2)
xml.student {
  xml.name("Jim")
  xml.phone_number("555-1234")
  xml.class("Intro to Ruby")
}
puts xml.target!
```

```
demo.rb:28:in `class': wrong number of
arguments (1 for 0) (ArgumentError)
    from demo.rb:28
    from demo.rb:12:in `method_missing'
    from demo.rb:25
```

The `class` method is predefined

How to Inherit from Object

Without inheriting from Object

?

Rather than Inherit from Object

```
class XmlBuilder
  def method_missing(sym, *args, &block)
    ...
  end
end
```

Inherit from BlankSlate

```
class XmlBuilder < BlankSlate
  def method_missing(sym, *args, &block)
    ...
  end
end
```

Blank Slate

```
class BlankSlate
  instance_methods.each do |name|
    undef_method name
  end
end
```

```
demo.rb:7: warning: undefining `__id__' may cause serious problem
demo.rb:7: warning: undefining `__send__' may cause serious problem
<student>
  <name>Jim</name>
  <phone_number>555-1234</phone_number>
  <class>Intro to Ruby</class>
</student>
```

Blank Slate

```
class BlankSlate
  instance_methods.each do |name|
    undef_method name unless name =~ /^__/
  end
end
```

```
<student>
  <name>Jim</name>
  <phone_number>555-1234</phone_number>
  <class>Intro to Ruby</class>
</student>
```

Good Enough?

Open Classes

```
require ‘blank_slate’

module Kernel
  def name
    “My Name”
  end
end
...
xml.name(“Jim”)
```

```
demo.rb:36:in `name': wrong number of arguments (1 for 0)
(ArgumentError)
```

First ... a Slight Rewrite

```ruby
class BlankSlate
  def self.hide(method)
    undef_method method
  end
  instance_methods.each do |name|
    hide(name) unless name =~ /^__/
  end
end
```

Catch New Methods

```
module Kernel
  class << self
    alias_method :original_method_added,
      :method_added

    def method_added(name)
      result = original_method_added(name)
      BlankSlate.hide(name) if self == Kernel
      result
    end
  end
end
```

Need Similar code for Object

Good Enough Now?

Not Quite

```
require ‘blank_slate’

module Name
  def name
    “My Name”
  end
end

class Object
  include Name
end
...
xml.name(“jim”)
```

```
demo.rb:36:in `name': wrong number of arguments (1 for 0)
(ArgumentError)
```

Solution

- Details are left to the student
- Hint: Use **`append_features`**
  - (instead of **`method_added`**)
- Bigger Hint: Look at BlankSlate in Builder

Box 3
Parsing without Parsing

Consider

```
User.find(:all,
  :conditions =>
     [“name = ?”, “jim”])
```

Consider

```
User.find(:all,
  :conditions =>
     [“name = ?”, “jim”])
```

vs

```
user_list.select { |user|
  user.name = “jim”
}
```

Wouldn’t it be nice if ...

we could use select on ActiveRecord models.

Like This

```
User.select { |user|
  user.name == “jim”
}
```

Naive Implementation

```ruby
class User
  def self.select(&block)
    find(:all).select(&block)
  end
end
```

What’s Wrong?

- Incredibly inefficient
    - Large tables will kill you
- Doesn’t take advantage of the database
- Did I mention it was inefficient?

Magic Implementation

```
class User
  def self.select(&block)
    cond =
      translate_block_to_sql(&block)
    find(:all, :conditions => cond)
  end
end
```

Magic Implementation

```
class User
  def self.select(&block)
    cond =
      translate_block_to_sql(&block)
    find(:all, :conditions => cond)
  end
end
```

How to Implement Magic?

(I) Parse the Source File

(II) ...

(III) ...

Parsing ... Ick!

```
expr     : command_call
         | expr kAND expr
             {
             $$ = logop(NODE_AND, $1, $3);
             }
         | expr kOR expr
             {
             $$ = logop(NODE_OR, $1, $3);
             }
         | kNOT expr
             {
             $$ = NEW_NOT(cond($2));
             }
         | '!' command_call
             {
             $$ = NEW_NOT(cond($2));
             }
         | arg
         ;
```

```
expr_value   : expr
                 {
                 value_expr($$);
                 $$ = $1;
                 }
             ;

command_call : command
             | block_command
             | kRETURN call_args
                 {
                 $$ = NEW_RETURN(ret_args($2));
                 }
             | kBREAK call_args
                 {
                 $$ = NEW_BREAK(ret_args($2));
                 }
             | kNEXT call_args
                 {
                 $$ = NEW_NEXT(ret_args($2));
                 }
             ;
```

How to Implement Magic?

(I) Parse the Source File

(II) Use Parse Tree

(III) ...

ParseTree ... Excellent Idea!

See **Ambition** for more details

http://errtheblog.com/post/10722

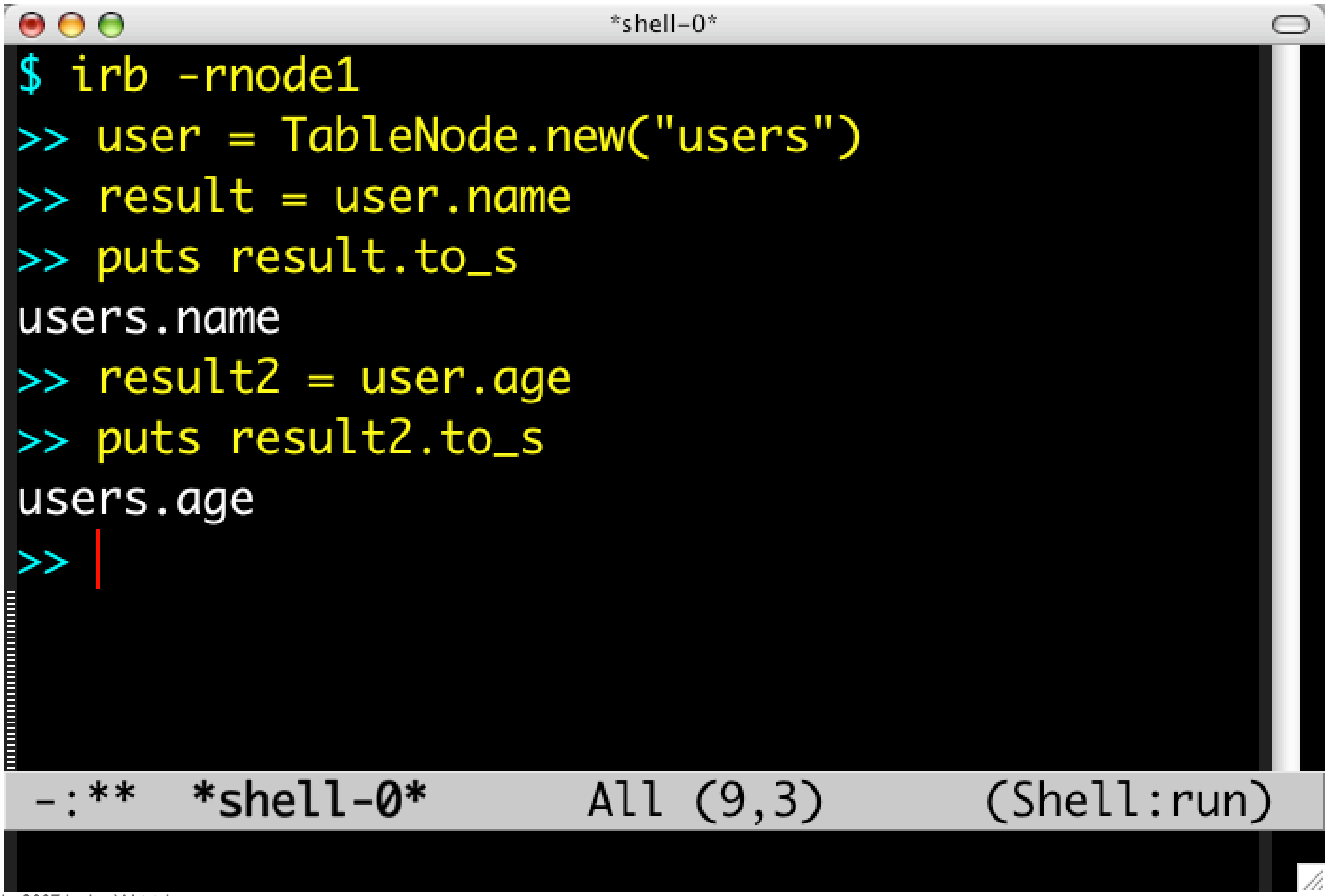
How to Implement Magic?

(I) Parse the Source File

(II) Use Parse Tree

(III) Just Execute the Code

Table Node



```
$ irb -rnode1
>> user = TableNode.new("users")
>> result = user.name
>> puts result.to_s
users.name
>> result2 = user.age
>> puts result2.to_s
users.age
>> 
```

```
-:**  *shell-0*       All (9,3)       (Shell:run)
```

Table Node

```ruby
class TableNode < Node
  def initialize(table_name)
    @table_name = table_name
  end

  def method_missing(sym, *args, &block)
    MethodNode.new(self, sym)
  end

  def to_s
    @table_name
  end
end
```

Method Node

```
class MethodNode < Node
  def initialize(obj, method)
    @obj = obj
    @method = method
  end

  def to_s
    "#{@obj}.#{@method}"
  end
end
```

How do we handle ...

```
User.select { |user|
  user.name == “jim”
}
```

Node

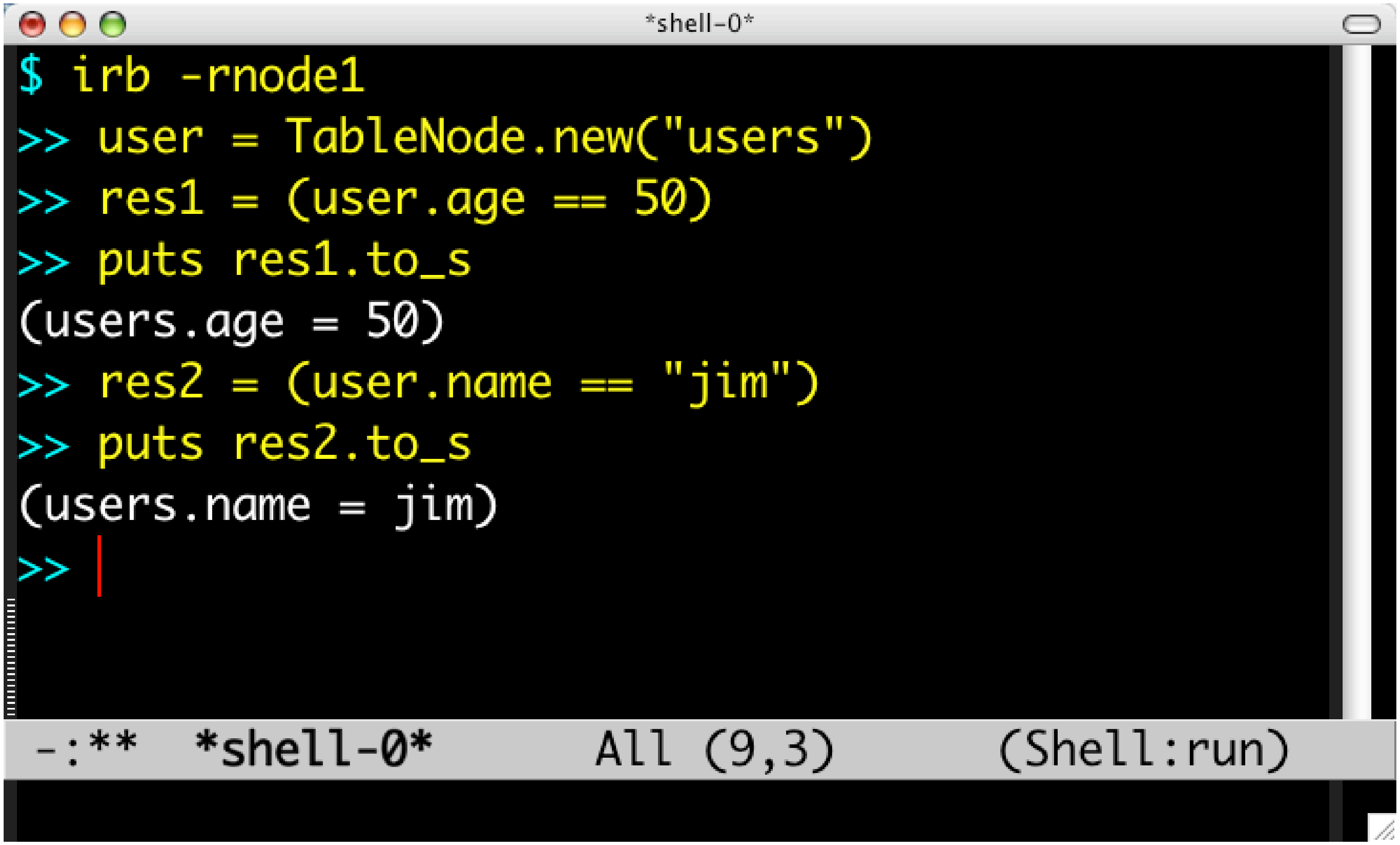
```ruby
class Node
  def ==(other)
    BinaryOpNode.new("=", self, other)
  end
end
```

BinaryOpNode

```
class BinaryOpNode < Node
  def initialize(operator, left, right)
    @operator = operator
    @left = left
    @right = right
  end

  def to_s
    "(#{@left} #{@operator} #{@right})"
  end
end
```

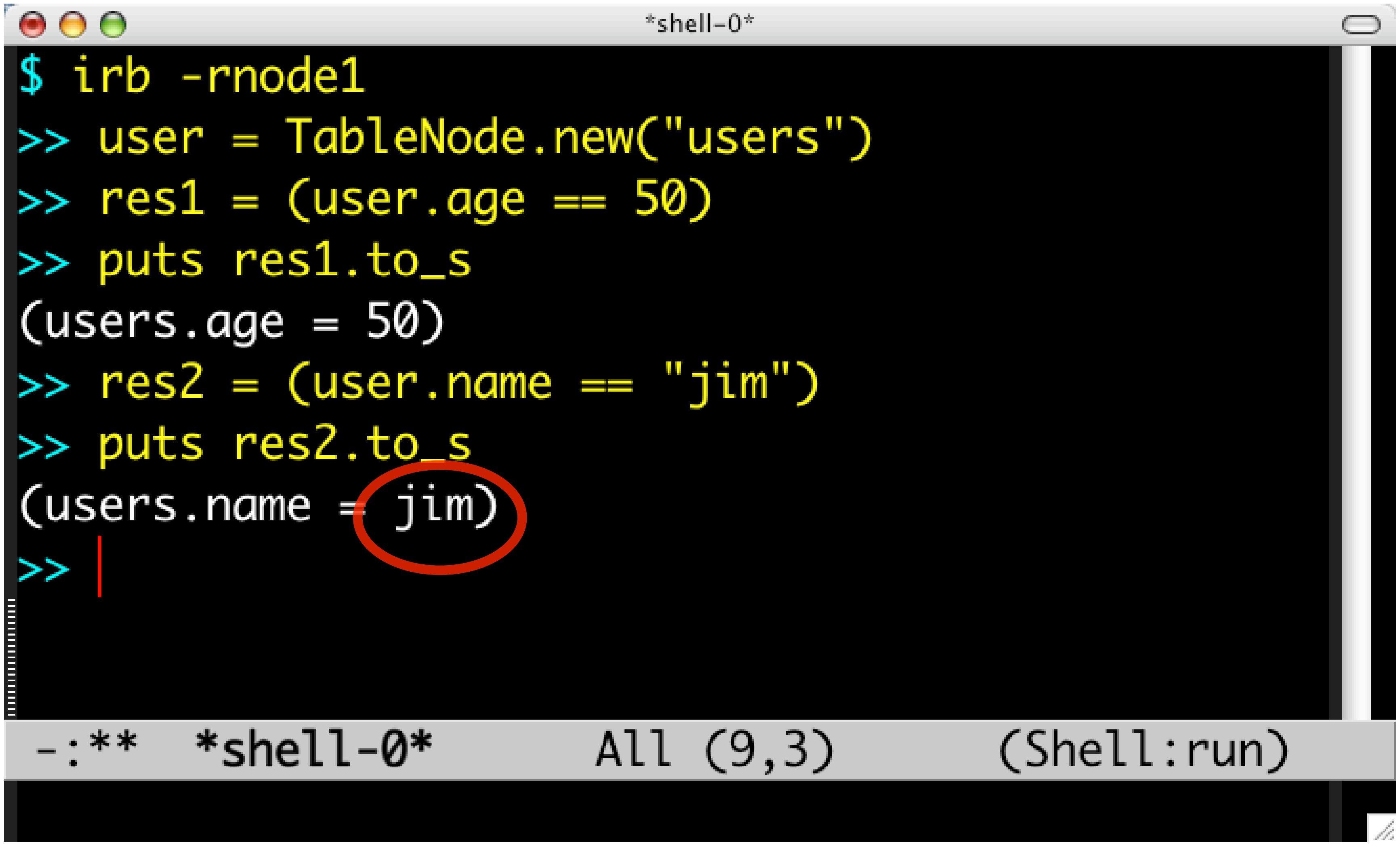
What Works So Far ...



```
$ irb -rnode1
>> user = TableNode.new("users")
>> res1 = (user.age == 50)
>> puts res1.to_s
(users.age = 50)
>> res2 = (user.name == "jim")
>> puts res2.to_s
(users.name = jim)
>> 
```

```
-:**  *shell-0*        All (9,3)        (Shell:run)
```

Where are the Quotes?


shell-0

```
$ irb -rnode1
>> user = TableNode.new("users")
>> res1 = (user.age == 50)
>> puts res1.to_s
(users.age = 50)
>> res2 = (user.name == "jim")
>> puts res2.to_s
(users.name = jim)
>> 
```

```
-:**  *shell-0*        All (9,3)        (Shell:run)
```

Some New Nodes

```
class LiteralNode
  def initialize(obj)
    @obj = obj
  end
  def to_s
    @obj.to_s
  end
end
```

```
class StringNode
  def initialize(string)
    @string = string
  end
  def to_s
    "'#{@string}'"
  end
end
```

We need a way to find the right node type for any object ...

Case Statement?

```ruby
def wrap_in_node(obj)
  case obj
  when String
    StringNode.new(obj)
  else
    LiteralNode.new(obj)
  end
end
```

Don't You Love Open Classes

```ruby
class Object
  def as_a_sql_node
    LiteralNode.new(self)
  end
end
```

```ruby
class String
  def as_a_sql_node
    StringNode.new(self)
  end
end
```

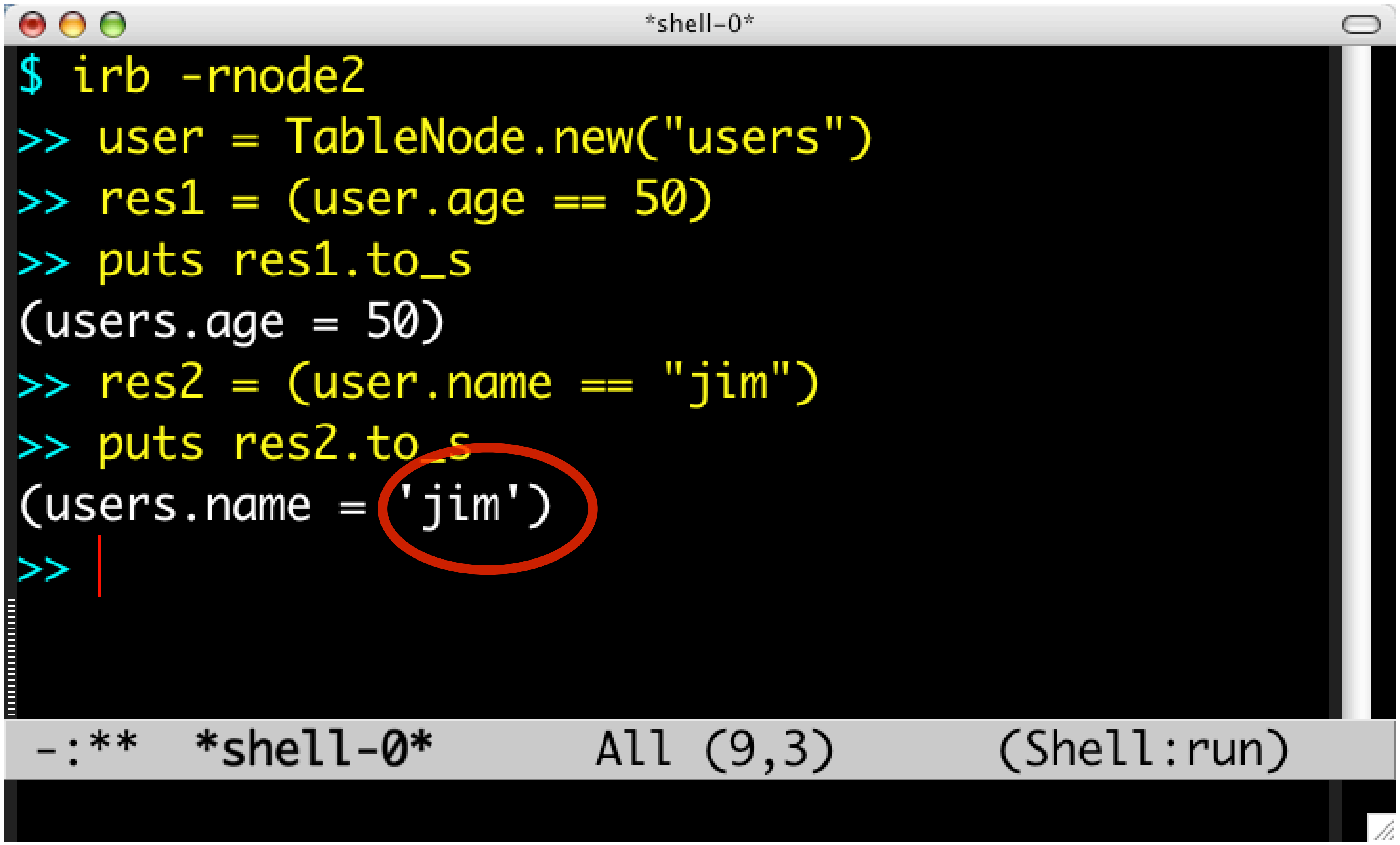
Some Tweeks to Node

```
class Node
  def ==(other)
    BinaryOpNode.new("=",
      self, other.as_a_sql_node)
  end
  def as_a_sql_node
    self
  end
end
```

Some Tweeks to Node

```
class Node
  def ==(other)
    BinaryOpNode.new("=",
      self, other.as_a_sql_node)
  end
  def as_a_sql_node
    self
  end
end
```

Quotes Look Good!



```
$ irb -rnode2
>> user = TableNode.new("users")
>> res1 = (user.age == 50)
>> puts res1.to_s
(users.age = 50)
>> res2 = (user.name == "jim")
>> puts res2.to_s
(users.name = 'jim')
>> 
-:**  *shell-0*       All (9,3)      (Shell:run)
```

What's Left To Do?

Other Operators

```
class Node
  def ==(other) ... end
  def <(other)  ... end
  def <=(other) ... end
  def +(other)  ... end
  def -(other)  ... end
  def *(other)  ... end
  def /(other)  ... end
  ...
end
```

Writing select

```
class User
  def self.select(&block)
    cond = block.call(
      TableNode.new(self.table_name))
    find(:all, :conditions => cond)
  end
end
```

Problems

Minor Problem

- Most operators are commutative

```
User.select { |user|
  user.name == “jim”
}
```

Minor Problem

- Literals on the left side might cause problems

```
User.select { |user|
  “jim” == user.name
}
```

- `coerce` can handle numeric operators.

Bigger Problem

- && and || can not be overridden in Ruby
    - They have short-circuit semantics
    - Cannot be implemented in a method
- Perhaps use & and | instead
    - but that breaks the paradigm we were striving for

Bigger Problem

- ! and != have predefined semantics in Ruby
  - You cannot change their meaning
  - You cannot override them

Prior Art

- The GLORP Smalltalk library provided inspirations for the dynamic parsing ideas.
- The Ruby "Criteria" library by Ryan Pavlik implemented many of these ideas.

Summary

What did we learn?

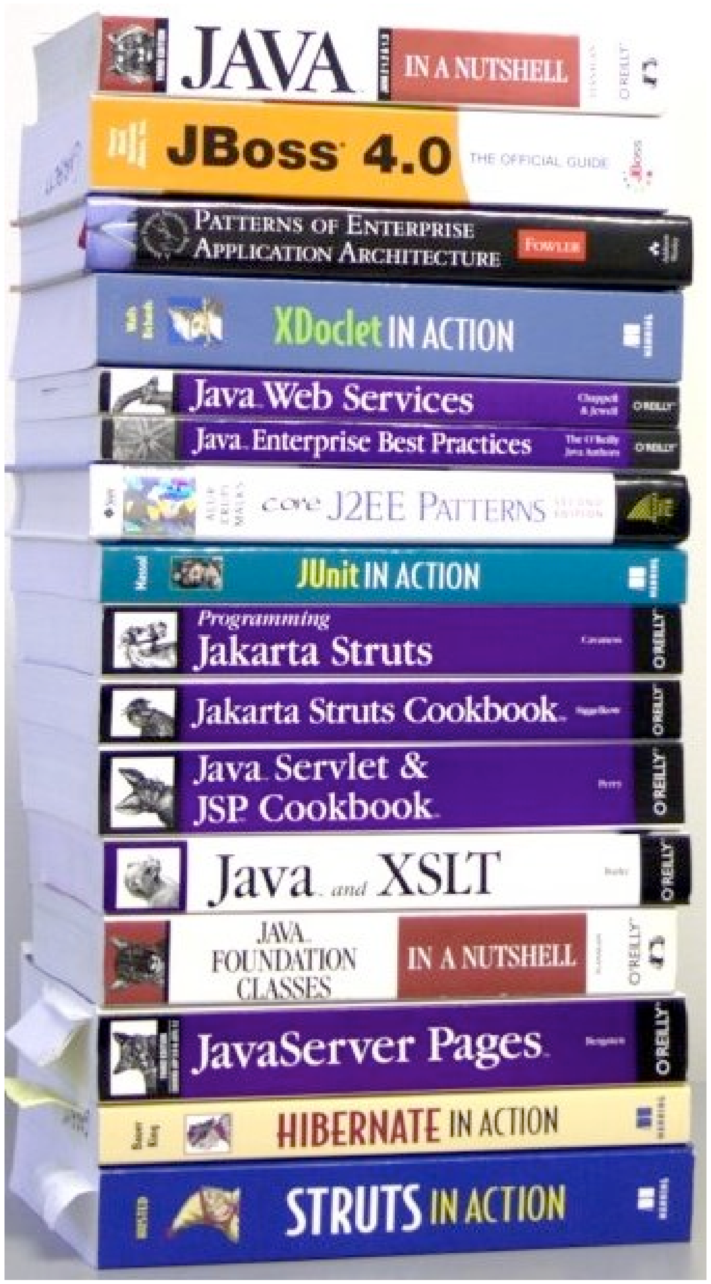
Programming Languages really do shape the way we solve problems.

Learn the corners of your language of choice to take full advantage

Don’t be afraid to think outside the box of past experience...

After all, if someone hadn't thought outside the box 3 years ago ...

I would still be programming in this:


JAVA IN A NUTSHELL
JBoss 4.0 THE OFFICIAL GUIDE
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
XDoclet IN ACTION
Java Web Services
Java Enterprise Best Practices
core J2EE PATTERNS
JUnit IN ACTION
Programming Jakarta Struts
Jakarta Struts Cookbook
Java Servlet & JSP Cookbook
Java and XSLT
JAVA FOUNDATION CLASSES IN A NUTSHELL
JavaServer Pages
HIBERNATE IN ACTION
STRUTS IN ACTION

Thank You

License