RARE AND OBSCURE RUBY

by Jonathan Arnett (J3RN)

RUBY 2 KEYWORD ARGUMENTS

NOT USING KEYWORD ARGUMENTS

```ruby
def foo(options = {})
  bar = options.fetch(:bar, 'default')
  puts bar
end

foo
# default
foo(bar: 'baz')
# baz
```

REAL WORLD EXAMPLE

(Tmuxinator)

KEYWORD ARGUMENTS

```ruby
def foo(bar: 'default')
  puts bar
end

foo
# default
foo(bar: 'baz')
# baz
```

RUBY 2.1 REQUIRED KEYWORD ARGUMENTS

```ruby
def foo(bar:)
  puts bar
end

foo # => ArgumentError: missing keyword: bar
foo(bar: 'baz')
# baz
```

KEYWORD ARGUMENTS ARE GOOD!

Use them in your code.

BLOCKS, PROCS, AND LAMBDAS

BLOCKS

```
[1, 2, 3].map { |x| x ** 2 }

test "anything makes sense these days" do
  assert true == true
end
```

BLOCKS

```
[1, 2, 3].map { |x| x ** 2 }

test "anything makes sense these days" do
  assert true == true
end
```

PROCS

```
square = Proc.new { |x| x ** 2 }
square.call(5)       #=> 25
```

BLOCKS

```
[1, 2, 3].map { |x| x ** 2 }

test "anything makes sense these days" do
  assert true == true
end
```

PROCS

```
square = Proc.new { |x| x ** 2 }
square.call(5)       #=> 25
```

LAMBDAS

```
square = lambda { |x| x ** 2 }
square.call(5)       #=> 25
```

THE PLOT THICKENS...

BLOCKS AND PROCS DON'T CHECK ARITY, LAMBDAS DO

BLOCKS AND PROCS DON'T CHECK ARITY, LAMBDAS DO

```
# Block
[1, 2, 3, 4].map { |x, y, z| x ** 2 } #=> [1, 4, 9, 16]
```

BLOCKS AND PROCS DON'T CHECK ARITY, LAMBDAS DO

```ruby
# Block
[1, 2, 3, 4].map { |x, y, z| x ** 2 } #=> [1, 4, 9, 16]
```

```ruby
# Proc
Proc.new { |x| x ** 2 }.call(2, 3, 4) #=> 4
```

BLOCKS AND PROCS DON'T CHECK ARITY, LAMBDAS DO

```ruby
# Block
[1, 2, 3, 4].map { |x, y, z| x ** 2 } #=> [1, 4, 9, 16]
```

```ruby
# Proc
Proc.new { |x| x ** 2 }.call(2, 3, 4) #=> 4
```

```ruby
# Lambda
lambda { |x| x ** 2 }.call(2, 3, 4)   #=> ArgumentError: wrong
```

BLOCKS AND PROCS DON'T CHECK ARITY, LAMBDAS DO

```ruby
# Block
[1, 2, 3, 4].map { |x, y, z| x ** 2 } #=> [1, 4, 9, 16]
```

```ruby
# Proc
Proc.new { |x| x ** 2 }.call(2, 3, 4) #=> 4
```

```ruby
# Lambda
lambda { |x| x ** 2 }.call(2, 3, 4)   #=> ArgumentError: wrong
```

Arity checking is good. Use lambdas.

BLOCK TO PROC CONVERSION

```ruby
def foo(&block)
  block.inspect
end

foo {} #=> #<Proc:0x007fc0fb107390@(pry):59>
```

BLOCK TO PROC CONVERSION

```ruby
def foo(&block)
  block.inspect
end

foo {} #=> #<Proc:0x007fc0fb107390@(pry):59>
```

This is a thing that you can do.

BLOCK TO PROC CONVERSION

```ruby
def foo(&block)
  block.inspect
end

foo {} #=> #<Proc:0x007fc0fb107390@(pry):59>
```

This is a thing that you can do.

It is physically possible to do this in your code.

PROC/LAMBDA CLOSURES

```ruby
def raise_to_power(power)
  lambda { |base| base ** power }
end

cube = raise_to_power(3)
cube.call(4) #=> 64
```

PROC/LAMBDA CLOSURES

```ruby
def raise_to_power(power)
  lambda { |base| base ** power }
end

cube = raise_to_power(3)
cube.call(4) #=> 64
```

Why would you use this in your code?

PROC METHOD RETURNS

```ruby
def foo
  Proc.new { return }.call
  "Hello, world!"
end

foo #=> nil
```

PROC METHOD RETURNS

```
def foo
  Proc.new { return }.call
  "Hello, world!"
end

foo #=> nil
```

This only works for procs.

PROC METHOD RETURNS

```
def foo
  Proc.new { return }.call
  "Hello, world!"
end

foo #=> nil
```

This only works for procs.

Please don't use this in your code.

THE : : NAMESPACE RESOLUTION OPERATOR

```ruby
class Foobar; end

module Barbaz
  class Foobar; end

  Foobar   #=> Barbaz::Foobar
  ::Foobar #=> Foobar
end
```

THE PLOT DOESN'T THICKEN

THE PLOT DOESN'T THICKEN

I'm just curious what caused you to need this.

CASE EQUALITY

===

GENERAL EXAMPLE

```ruby
class Foo
  def ===(obj)
    obj == 1
  end
end

foo = Foo.new
foo === 1 #=> true
foo == 1  #=> false
```

CASE STATEMENT

```
case 1
when foo
  puts "Everything is weird"
else
  puts "Everything is broken"
end
```

CASE STATEMENT

```
case 1
when foo
  puts "Everything is weird"
else
  puts "Everything is broken"
end
```

```
# Everything is weird
```

THE PLOT THICKENS

THE PLOT THICKENS

- Order is important.

```
(1 === foo) != (foo === 1)
```

THE PLOT THICKENS

- Order is important.

```
(1 === foo) != (foo === 1)
```

- Especially here!

```
case foo
when 1
  # Never reached
end
```

THE PLOT THICKENS

- Order is important.

```
(1 === foo) != (foo === 1)
```

- Especially here!

```ruby
case foo
when 1
  # Never reached
end
```

- In Ruby, there's `equal?`, `eql?`, `==`, and `===`. They all do different things. This will, one day, drive someone insane. Make sure it's not you.

HASH EQUALITY

.EQL?

THIS METHOD IS DUMB

THIS METHOD IS DUMB

- This method should return `true` if the two objects have the same `hash`.

THIS METHOD IS DUMB

- This method should return `true` if the two objects have the same `hash`.
- There is no programatic reason why that is true. It sometimes is not true. It is a worthless distinction.

THIS METHOD IS DUMB

- This method should return `true` if the two objects have the same `hash`.
- There is no programatic reason why that is true. It sometimes is not true. It is a worthless distinction.
- `Object#eql?` is just an alias to `Object#==`, and this behavior is widely used.

THIS METHOD IS DUMB

- This method should return `true` if the two objects have the same `hash`.
- There is no programatic reason why that is true. It sometimes is not true. It is a worthless distinction.
- `Object#eql?` is just an alias to `Object#==`, and this behavior is widely used.
- The primary exceptions are `Numeric` types, in which == does type conversion, but `eql?` does not.

THIS METHOD IS DUMB

- This method should return `true` if the two objects have the same `hash`.
- There is no programatic reason why that is true. It sometimes is not true. It is a worthless distinction.
- `Object#eql?` is just an alias to `Object#==`, and this behavior is widely used.
- The primary exceptions are `Numeric` types, in which == does type conversion, but `eql?` does not.

No.

THIS METHOD IS DUMB

- This method should return `true` if the two objects have the same `hash`.
- There is no programatic reason why that is true. It sometimes is not true. It is a worthless distinction.
- `Object#eql?` is just an alias to `Object#==`, and this behavior is widely used.
- The primary exceptions are `Numeric` types, in which == does type conversion, but `eql?` does not.

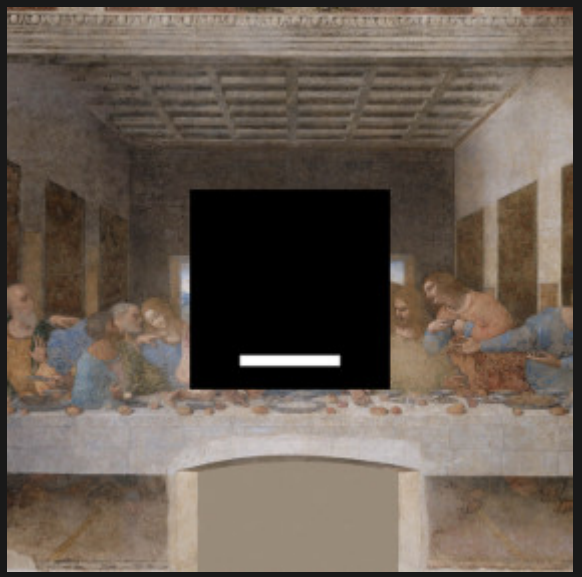
No. I don't think so.

THIS METHOD IS DUMB

- This method should return `true` if the two objects have the same `hash`.
- There is no programatic reason why that is true. It sometimes is not true. It is a worthless distinction.
- `Object#eql?` is just an alias to `Object#==`, and this behavior is widely used.
- The primary exceptions are `Numeric` types, in which == does type conversion, but `eql?` does not.

No. I don't think so. Let's not.

THE LAST VALUE



```
Person.last #=> #<Person id: 5894674, name-last: "Nunez" ...>
```

```
Person.last #=> #<Person id: 5894674, name-last: "Nunez" ...>

nunez = _   #=> #<Person id: 5894674, name-last: "Nunez" ...>
```

THE PLOT THICKENS

THE PLOT THICKENS

- This behavior only exists in interactive shells.

THE PLOT THICKENS

- This behavior only exists in interactive shells.
- In files, _ is just a normal variable.

THE PLOT THICKENS

- This behavior only exists in interactive shells.
- In files, _ is just a normal variable.
- *Don't use _ as a variable in your code*

METHOD MISSING

```ruby
class Foobar
  def method_missing(name)
    name.to_s
  end
end

foo = Foobar.new
foo.supercalifragilisticexpialidocious #=> "supercalifragilist
```

THE PLOT THICKENS

THE PLOT THICKENS

- `respond_to` does not understand `method_missing`.

```
foo.respond_to?(:supercalifragilisticexpialidocious) #=> fa
```

THE PLOT THICKENS

- `respond_to` does not understand `method_missing`.

```
foo.respond_to?(:supercalifragilisticexpialidocious) #=> fa
```

- A developer searching for

```
def supercalifragilisticexpialidocious
```

will never find it.

THE PLOT THICKENS

- `respond_to` does not understand `method_missing`.

```
foo.respond_to?(:supercalifragilisticexpialidocious) #=> fa
```

- A developer searching for

```
def supercalifragilisticexpialidocious
```

  will never find it.
- Try not to use `method_missing` in your code.

NIL PIPE

THE "TRUTHINESS" OPERATOR

THE "TRUTHINESS" OPERATOR

```
nil | "false" #=> true
nil | 0       #=> true
nil | nil     #=> false
nil | false   #=> false
```

THE PLOT GETS NO THICKER

THE PLOT GETS NO THICKER

Why would you use this in your code?

Thanks to Hakim El Hattab for his work on reveal.js!