

# Ruby: Another Gem of a Language

---

Simon Cozens

# What does Ruby look like?

```
class Complex < Numeric
```

```
  def initialize(a, b = 0)
```

```
    @real = a
```

```
    @image = b
```

```
  end
```

```
  def Complex.generic?(other)
```

```
    other.kind_of?(Integer) or
```

```
    other.kind_of?(Float) or
```

```
    (defined?(Rational) and other.kind_of?(Rational))
```

```
  end
```

# What does Ruby look like?

```
class Complex < Numeric X
```

Braces?

```
def initialize(a, b = 0) X
```

Semicolons?

```
  @real = a X
```

End of statement  
intuited

```
  @image = b X
```

```
X
```

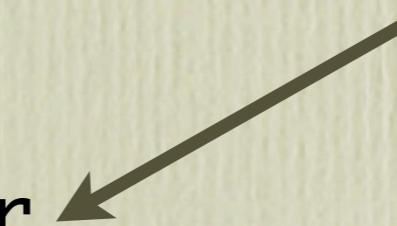
```
def Complex.generic?(other)
```

```
  other.kind_of?(Integer) or
```

```
  other.kind_of?(Float) or
```

```
  (defined?(Rational) and other.kind_of?(Rational))
```

```
end
```



# What does Ruby look like?

```
class Complex < Numeric
```

```
  def initialize(a, b = 0)
```

```
    @real = a
```

```
    @image = b
```

```
  end
```

```
  def Complex.generic?(other)
```

```
    other.kind_of?(Integer) or
```

```
    other.kind_of?(Float) or
```

```
    (defined?(Rational) and other.kind_of?(Rational))
```

```
  end
```

# What does Ruby look like?

```
class Complex < Numeric
  def initialize(a, b = 0)
    @real = a
    @image = b
  end

  def Complex.generic?(other)
    other.kind_of?(Integer) or
    other.kind_of?(Float) or
    (defined?(Rational) and other.kind_of?(Rational))
  end
```

Inheriting from a  
built-in class

# What does Ruby look like?

```
class Complex < Numeric  
  def initialize(a, b = 0)  
    @real = a  
    @image = b  
  end
```

Method arguments,  
with a default

```
def Complex.generic?(other)  
  other.kind_of?(Integer) or  
  other.kind_of?(Float) or  
  (defined?(Rational) and other.kind_of?(Rational))  
end
```

# What does Ruby look like?

```
class Complex < Numeric  
  def initialize(a, b = 0)  
    @real = a  
    @image = b  
  end
```

*object attributes*

Complex  
@real: 10  
@image: -5

```
def Complex.generic?(other)  
  other.kind_of?(Integer) or  
  other.kind_of?(Float) or  
  (defined?(Rational) and other.kind_of?(Rational))  
end
```

# What does Ruby look like?

```
class Complex < Numeric  
  def initialize(a, b = 0)  
    @real = a  
    @image = b  
  end
```

```
  def Complex.generic?(other)  
    other.kind_of?(Integer) or  
    other.kind_of?(Float) or  
    (defined?(Rational) and other.kind_of?(Rational))  
  end
```

*class method*

# What does Ruby look like?

```
class Complex < Numeric  
  def initialize(a, b = 0)  
    @real = a  
    @image = b  
  end
```

*Method call operator*

```
  def Complex.generic?(other)  
    other.kind_of?(Integer) or  
    other.kind_of?(Float) or  
    (defined?(Rational) and other.kind_of?(Rational))  
  end
```



# What does Ruby look like?

```
class Complex < Numeric  
  def initialize(a, b = 0)  
    @real = a  
    @image = b  
  end
```

```
  def Complex.generic?(other)  
    other.kind_of?(Integer) or  
    other.kind_of?(Float) or  
    (defined?(Rational) and other.kind_of?(Rational))  
  end
```

Introspection.

class names are  
first class objects

# What does Ruby look like?

```
def + (other)
  if other.kind_of?(Complex)
    re = @real + other.real
    im = @image + other.image
    Complex(re, im)
  elsif Complex.generic?(other)
    Complex(@real + other, @image)
  else
    x, y = other.coerce(self)
    x + y
  end
end
```

# What does Ruby look like?

```
def + (other)
  if other.kind_of?(Complex)
    re = @real + other.real
    im = @image + other.image
    Complex(re, im)
  elsif Complex.generic?(other)
    Complex(@real + other, @image)
  else
    x, y = other.coerce(self)
    x + y
  end
end
```

*Operator  
overloading*

Yes, operators are methods

`x = 2 + 3`

`x = 2.+ (3)`

# What does Ruby look like?

```
def + (other)
  if other.kind_of?(Complex)
    re = @real + other.real
    im = @image + other.image
    Complex(re, im)
  elsif Complex.generic?(other)
    Complex(@real + other, @image)
  else
    x, y = other.coerce(self)
    x + y
  end
end
```

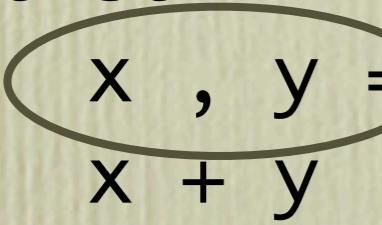
*Operator  
overloading*

# What does Ruby look like?

```
def + (other)
  if other.kind_of?(Complex)
    re = @real + other.real
    im = @image + other.image
    Complex(re, im)
  elsif Complex.generic?(other)
    Complex(@real + other, @image)
  else
    x, y = other.coerce(self)
    x + y
  end
end
```

*java-style  
constructor calls*

# What does Ruby look like?

```
def + (other)
  if other.kind_of?(Complex)
    re = @real + other.real
    im = @image + other.image
    Complex(re, im)
  elsif Complex.generic?(other)
    Complex(@real + other, @image)
  else
    
    x , y = other.coerce(self)
    x + y
  end
end
```

*List assignment*

# Where does Ruby come from?

- Japan
- Yukihiro Matsumoto

# Where does Ruby come from?

- Perl
- Python
- CLU
- Smalltalk

# Why Ruby?

- It's a good starting point
- It's Perl 6 (more or less)
- Languages are good for you!
- It's a lot of fun!

# Ease of Expression

- “If there are any elements in array A which also appear in array B, take them out of array A, and put them in array C”

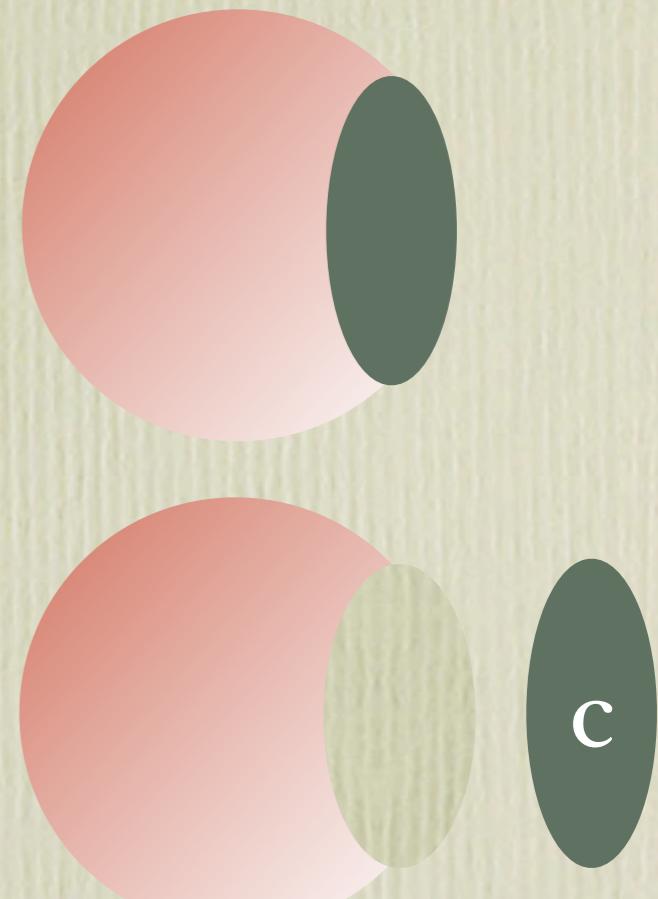
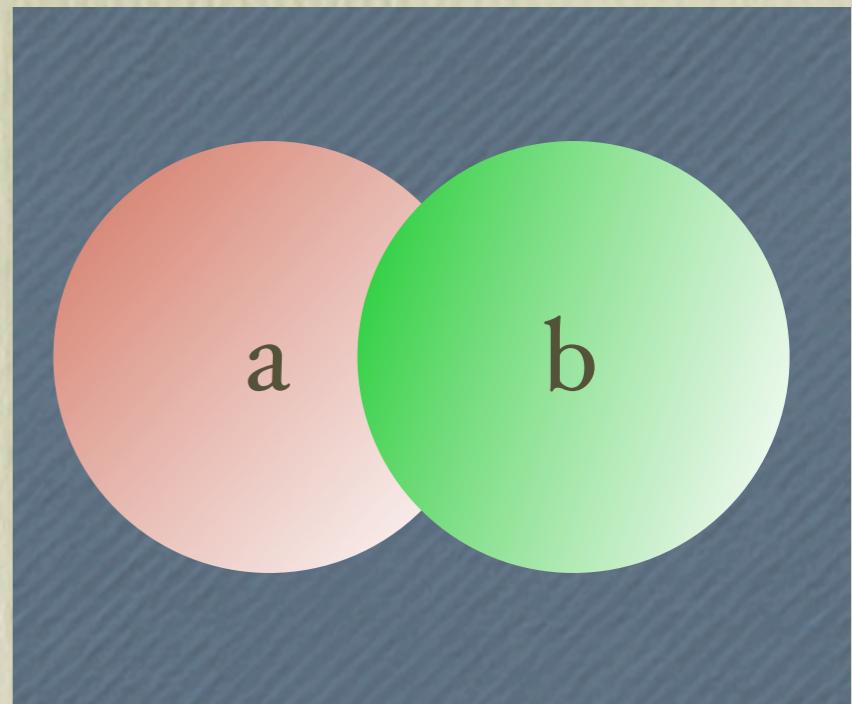
# Ease of Expression

```
my @intersect = _set_intersection(@a, @b);
if (@intersect) {
    @a = _set_difference(\@a, \@intersect);
    push @c, @intersect;
}
```

# Perl Cookbook, chapter 4...

```
sub _set_intersection {
    my %union; my %isect;
    for (@_) { $union{$_}++ && ($isect{$_}=$_) }
    return values %isect;
}
```

```
sub _set_difference {
    my ($a, $b) = @_;
    my %seen; @seen{@$b} = ();
    return grep { !exists $seen{$_} } @$a;
}
```



# Ease of Expression

```
intersect = a & b  
if (intersect.length > 0)  
    a -= intersect  
    c += intersect  
end
```

If there are any elements in array A which also appear in array B take them out of array A and put them in array C

# Structure of Language

- Operators
- Control structures
- Built-in methods

# Concepts of Equality

a  
↓

Person  
Age: 19  
Name: Sarah

=  
?

Person  
Age: 19  
Name: Sarah

b  
↓

# Concepts of Equality

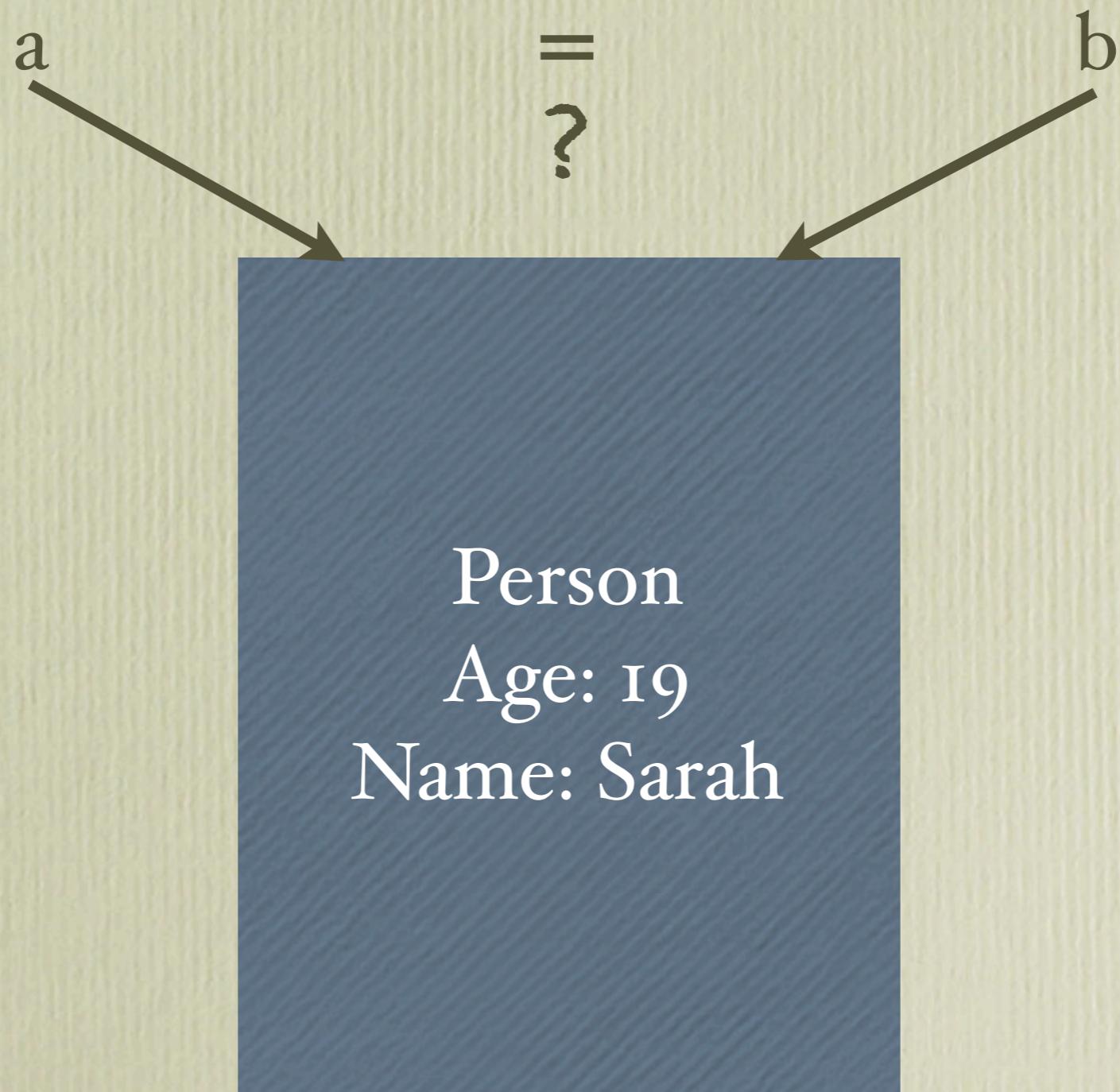
a  
↓

Person  
Age: 19  
Name: Sarah

=  
?

"Sarah"

# Concepts of Equality



# Concepts of Equality

- Equality of data: `a == b`
- Equality of concept: `a === "Sarah"`
- Equality of reference: `a.equal?(b)`
- (Match: `a =~ b`)

# String Operators

- % : Format operator
  - "%s: %2.02f GBP" % ["Coffee", 3]
- \* : Repeat operator
  - "\n" \* 3

# String Operators

- + / << : Catenation operator
- STDOUT << "I love Ruby" << "And C++"
- []: Indexing operator
  - "foo bar"[3] # 32

# Pushing onto an Array

```
irb(main):001:0> a = [1,2,3]; b = [3,4,5]; a + b  
=> [1, 2, 3, 3, 4, 5]  
irb(main):002:0> a << 4  
=> [1, 2, 3, 4]  
irb(main):003:0> a = [1,2,3]; a << b  
=> [1, 2, 3, [3, 4, 5]]  
irb(main):004:0> a = [1,2,3]; a | b  
=> [1, 2, 3, 4, 5]  
irb(main):005:0> a.push(b)  
=> [1, 2, 3, [3, 4, 5]]  
irb(main):006:0> a.push(*b)  
=> [1, 2, 3, [3, 4, 5], 3, 4, 5]
```

# Defining our own

```
myhash = { "Ruby" => 10, "Perl" => 5, "Python" => 1 };  
myhash === "Perl"
```

```
class Hash  
  def ==(key)  
    return has_key?(key)  
  end  
end
```

# Ruby's Case Statement

```
case value
when String
  puts "Value is a string!"
when "Hello world!"
  puts "Value is a familiar greeting!"
when myhash
  puts "Value is the name of a programming
language"
end
```

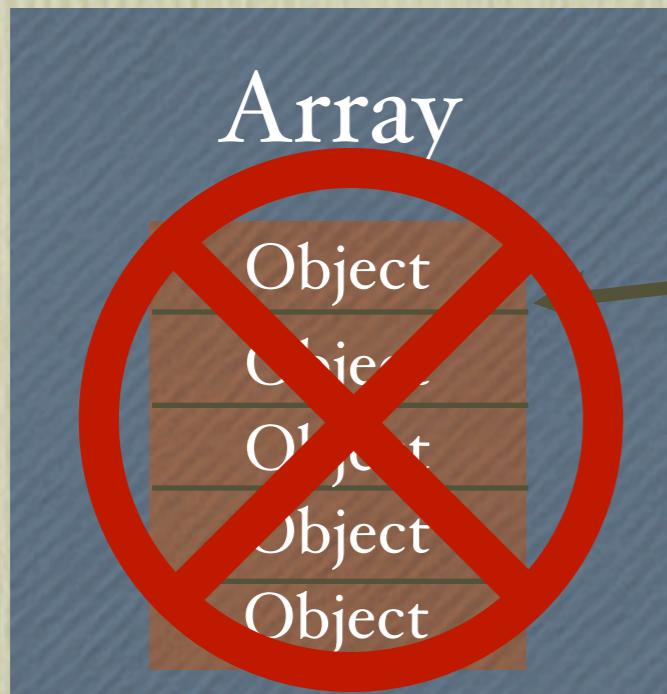
# Ruby's Case Statement

```
if String === value
  puts "Value is a string!"
elsif "Hello world!" === value
  puts "Value is a familiar greeting!"
elsif myhash === value
  puts "Value is the name of a programming
language"
end
```

Thing to be compared goes on the right

Argument to 'when' gets to decide how comparison is done

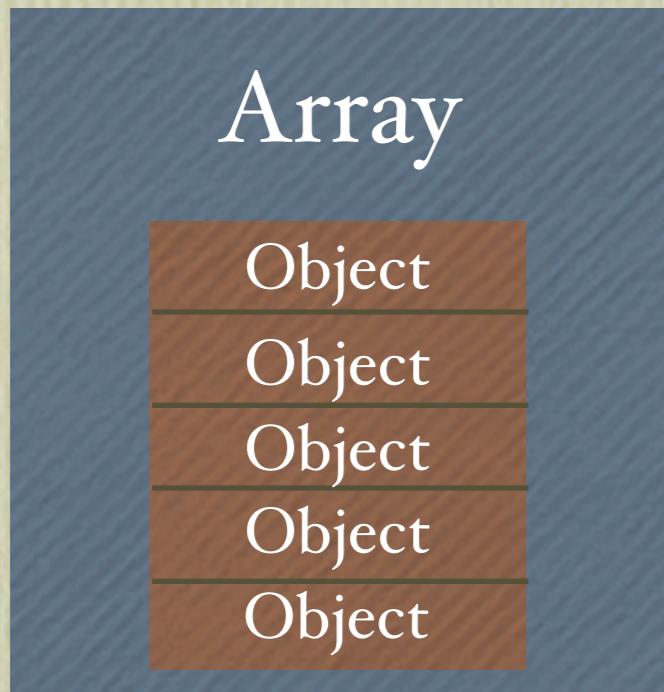
# Why Ruby cannot have a ‘for’ statement



Iterator walks the  
list

Breaks object abstraction - Bad!

# How Ruby manages to have a ‘for’ statement



array.each { |x| ... }

for x in array

...

end

# Iterating with ‘each’

```
irb(main):001:0> a = [ "Just", "another", "Ruby", "hacker" ]  
=> ["Just", "another", "Ruby", "hacker"]
```

```
irb(main):002:0> a.each { |string| print string, " " }  
Just another Ruby hacker => ["Just", "another", "Ruby", "hacker"]
```

# Parameters and Blocks

```
object.method  
( param1, param2, param3 )  
{  
| blockarg1, blockarg2 |  
...  
}
```

# Parameters and Blocks

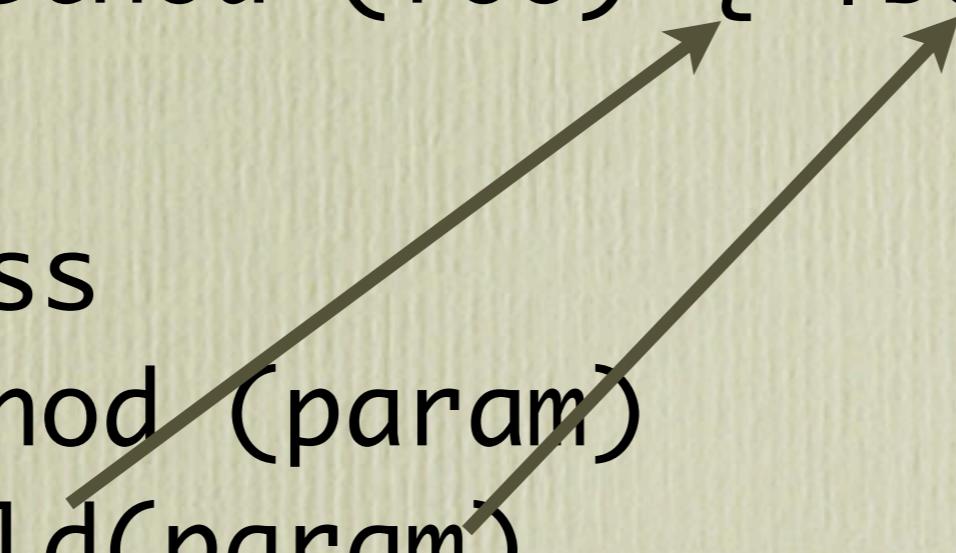
```
object.method (foo) { |bar| p bar }
```

```
class Klass
  def method (param)
    yield(param)
  end
end
```

# Parameters and Blocks

```
object.method (foo) { |bar| p bar }
```

```
class Klass
  def method(param)
    yield(param)
  end
end
```



The diagram illustrates the binding of parameters between a class method definition and a block. Two arrows point from the word 'param' in the class definition to the parameter 'param' in the block definition, indicating that they refer to the same variable.

# Creating our own iterators

```
class Array
  def each_backwards
    self.reverse.each{ yield(x) }
  end
end

[5, 10, 15, 20].each_backwards { |x| puts x}
```

20  
15  
10  
5

# Control Structure Miscellany

- ‘if’ has ‘unless’, ‘while’ has ‘until’
- Statements are expressions
- Statement modifiers
- Exceptions are caught with begin/rescue/end

# Looking into Methods

```
irb(main):003:0> String.instance_methods.sort
=> [%", "*", "+", "<<", "<=>", "==", "====", "=~", "[]",
"[]=", "capitalize", "capitalize!", "center", "chomp",
"chomp!", "chop", "chop!", "clone", "concat", "count",
"crypt", "delete", "delete!", "downcase", "downcase!",
"dump", "dup", "each", "each_byte", "each_line", "empty?",
"eql?", "gsub", "gsub!", "hash", "hex", "include?", "index",
"inspect", "intern", "length", "ljust", "next", "next!",
"oct", "replace", "reverse", "reverse!", "rindex", "rjust",
"scan", "size", "slice", "slice!", "split", "squeeze",
"squeeze!", "strip", "strip!", "sub", "sub!", "succ",
"succ!", "sum", "swapcase", "swapcase!", "to_f", "to_i",
"to_s", "to_str", "tr", "tr!", "tr_s", "tr_s!", "unpack",
"upcase", "upcase!", "upto", "~"]
```

# Substitution with blocks

# Built-in Methods for Everything

```
for (keys %attrs) {  
    delete $attrs{$_} unless defined $attrs{$_}  
}  
  
attrs.delete_if { |k,v| v.nil? }
```

# The Schwartzian Transform

```
@sorted = map { $_[0] }
            sort { $a->[1] cmp $b->[1] }
            map { [ $_[0], function($_) ] }
@array;

sorted = array
        .map { |x| [ x, function(x) ] }
        .sort { |a,b| a[1].cmp(b[1]) }
        .map { |x| x[0] }

sorted = array.sort_by { |x| function(x) }
```

# Iterators are Finalizers

```
def transaction
    sth = self.make_statement_handle();
    yield sth
    sth.commit!
end
```

```
database.transaction do |sth|
    sth.update(...)
end
```

# Checking POP Mail

```
require 'net/pop'  
pop = Net::POP3::new("pop.simon-cozens.org");  
pop.start("simon", password) do  
  pop.each { |mail| puts "Incoming mail", mail.header }  
end
```

# Ruby Resources

- “The Ruby Way”, Hal Fulton, Sams
- Why’s (Poignant) Guide to Ruby:  
<http://www.poignantguide.net/>
- These slides:  
<http://simon-cozens.org/programmer/talks/ruby-20040420.pdf>
- Tutorial notes for this talk:  
<http://simon-cozens.org/programmer/talks/ruby-20040420-tutorial.pdf>