Class
Proc < Object

Proc objects are blocks of code that have been bound to a set of local variables. Once bound, the code may be called in different contexts and still access those variables.

```
def gen_times(factor)
  return Proc.new {|n| n*factor }
end

times3 = gen_times(3)
times5 = gen_times(5)

times3.call(12)               # =>  36
times5.call(5)                # =>  25
times3.call(times5.call(4))   # =>  60
```

Class methods

new Proc.new { *block* } → *a_proc*
Proc.new → *a_proc*

Creates a new Proc object, bound to the current context. Proc.new may be called without a block only within a method with an attached block, in which case that block is converted to the Proc object.

```
def proc_from
  Proc.new
end
proc = proc_from { "hello" }
proc.call   # =>  "hello"
```

Instance methods

[ ] *prc*[⟨ *params* ⟩*] → *obj*

Synonym for Proc.call.

== *prc*== *other* → true or false

Returns true if *prc* is the same as *other*.

=== *prc*=== *other* → *obj*

1.9

Equivalent to *prc*.call(other). Allows you to use procs in when clauses. Allows us to write stuff such as this:

```
even = lambda {|num| num.even? }
(0..3).each do |num|
  case num
  when even
    puts "#{num} is even"
  else
    puts "#{num} is not even"
  end
end
```

produces:

```
0 is even
1 is not even
2 is even
3 is not even
```

arity

prc.arity → *integer*

Returns the number of arguments required by the block. If the block is declared to take no arguments, returns 0. If the block is known to take exactly n arguments, returns n. If the block has optional arguments, return $-(n+1)$, where n is the number of mandatory arguments. A proc with no argument declarations also returns -1, because it can accept (and ignore) an arbitrary number of parameters.

```
Proc.new {}.arity          # =>  0
Proc.new {||}.arity        # =>  0
Proc.new {|a|}.arity       # =>  1
Proc.new {|a,b|}.arity     # =>  2
Proc.new {|a,b,c|}.arity   # =>  3
Proc.new {|*a|}.arity      # =>  -1
Proc.new {|a,*b|}.arity    # =>  -2
```

1.9 In Ruby 1.9, arity is defined as the number of parameters that would not be ignored. In 1.8, Proc.new{}.arity returns -1, and in 1.9 it returns 0.

call

prc.call(⟨ *params* ⟩*) → *obj*

Invokes the block, setting the block's parameters to the values in *params* using something close to method-calling semantics. Returns the value of the last expression evaluated in the block.

```
a_proc = Proc.new {|a, *b| b.collect {|i| i*a }}
a_proc.call(9, 1, 2, 3)   # =>   [9, 18, 27]
a_proc[9, 1, 2, 3]        # =>   [9, 18, 27]
```

1.9 If the block being called accepts a single parameter and you give call more than one parameter, only the first will be passed to the block. This is a change from Ruby 1.8.

```
a_proc = Proc.new {|a| puts a}
a_proc.call(1,2,3)
```

produces:

```
1
```

If you want a block to receive an arbitrary number of arguments, define it to accept *args.

```
a_proc = Proc.new {|*a| p a}
a_proc.call(1,2,3)
```

produces:

```
[1, 2, 3]
```

Blocks created using Kernel.lambda check that they are called with exactly the right number of parameters.

```
p_proc = Proc.new {|a,b| puts "Sum is: #{a + b}" }
p_proc.call(1,2,3)
p_proc = lambda {|a,b| puts "Sum is: #{a + b}" }
p_proc.call(1,2,3)
```

produces:

```
Sum is: 3
prog.rb:4:in `call': wrong number of arguments (3 for 2) (ArgumentError)
from /tmp/prog.rb:5:in `<main>'
```

curry

prc.curry → *curried_proc*

1.9

If you have a proc that takes arguments, you normally have to supply all of those arguments if you want the proc to execute successfully. However, it is also possible to consider an n argument proc to be the same as a single argument proc that returns a new proc that has this first argument fixed and that takes $n - 1$ arguments. If you repeat this process recursively for each of these subprocs, you end up with a proc that will take from zero to n arguements. If you pass it all n, it simply executes the proc with those arguments. If you pass it m arguments (where $m < n$), it returns a new proc that has those arguments prebaked in and that takes $m - n$ arguements. In this way, it is possible to partially apply arguments to a proc.

```
add_three_numbers = lambda {|a,b,c| a + b + c}
add_10_to_two_numbers = add_three_numbers.curry[10]
add_33_to_one_number  = add_10_to_two_numbers[23]

add_three_numbers[1,2,3]     # =>  6
add_10_to_two_numbers[1,2]   # =>  13
add_33_to_one_number[1]      # =>  34
```

lambda?

prc.lambda? → true or false

1.9

Returns true if *prc* has lambda semantics (that is, if argument passing acts as it does with method calls). See the discussion starting on page 363.

source_location

prc.source_location → [*filename*, *lineno*] or nil

1.9

Returns the source filename and line number where *prc* was defined or nil if self was not defined in Ruby source.

```
variable = 123
prc = lambda { "some proc" }
prc.source_location   # =>   ["/tmp/prog.rb", 2]
```

to_proc

prc.to_proc → *prc*

Part of the protocol for converting objects to Proc objects. Instances of class Proc simply return themselves.

to_s

prc.to_s → *string*

Returns a description of *prc*, including information on where it was defined.

```
def create_proc
  Proc.new
end

my_proc = create_proc { "hello" }
my_proc.to_s   # =>   "#<Proc:0x001c7abc@prog.rb:5>"
```

yield

prc.yield(⟨ *params* ⟩*) → *obj*

1.9

Synonym for Proc#call.