Method

Defining a method:

Methods are defined with the def keyword, followed by the method name and an optional list of parameter names in parentheses. The Ruby code between def and end represents the body of the method.

```ruby
def hello(name)
 "Hello, #{name}"
end
```

A method invocation specifies the method name, the object on which it is to be invoked (sometimes called the receiver), and zero or more argument values that are assigned to the named method parameters.

```ruby
hello("World")
# => "Hello, World"
```

When the receiver is not explicit, it is self.

Because Ruby is a purely object-oriented language, all methods are true methods and are associated with at least one object.

```ruby
self
# => main

self.hello("World")
# => "Hello, World"
```

We do not declare the return type; a method returns the value of the last statement executed in the method. The parentheses around a method's arguments are optional. When a method has arguments and omit them when it doesn't. In Rails, you will see methods calls with no parentheses.

```ruby
def test
 'Hello'
end

puts test

result:Hello
       => nil


# Method with an argument - arity=1
def test1(name)
 'Hello ' + name
end
puts(test1('nuts'))

result: Hello nuts
        => nil


def test2 name2
 'Hello ' + name2
end
puts(test2 'nuts')

result:Hello nuts
       => nil
```

Default parameters:

Ruby lets you specify default values for a method's arguments-values that will be used if the caller doesn't pass them explicitly. Parameter defaults are evaluated when a method is invoked rather than when it is parsed.

```ruby
def method(arg1="nuts", arg2="neil")
   "#{arg1} #{arg2}"
end
#String interpolation calls to_s method on object and returns the string
value
```

```
puts method
result: nuts neil


puts method("daniel")
result: daniel neil
```

However, it's not possible to supply the second without also supplying the first. Instead of using positional parameters, try keyword parameters:

keyword parameters:

```
puts method(arg1: "daniel") # Duck typing, A hash without braces
is sometimes called a bare hash

result: daniel neil
```

Splat Operator:

```
def welcome_guests(*guests)
   guests.each { |guest| puts "Welcome #{guest}!" }
end
welcome_guests('Tom')
Welcome Tom!
=> ["Tom"]
welcome_guests('Tom', 'Tim', 'Lucas')
Welcome Tom!
Welcome Tim!
Welcome Lucas!
=> ["Tom", "Tim", "Lucas"]
```

Aliasing Method:

alias test_method method

When a method is aliased, the new name refers to a copy of the original method's body

```
def oldMethod
 "old method definition"
end
alias newMethod oldMethod
```

```
def oldMethod
 "old method improvised"
end
puts oldMethod
result:
old method improvised
=> nil
puts newMethod
result:
old method definition
        => nil
```

Methods Evaluation:

1. `meth == other_meth → true or false`
   `eql?(other_meth) → true or false`

Two method objects are equal if they are bound to the same object and refer to the same method definition and their owners are the same class or module.

```
def method1
  puts "testing....."
end


def method2
  puts "hello......"
  puts "testing....."
end

method1 == method2
result : true

method1.object_id == method2.object_id
result : true
```

2. `meth[args, ...] → obj click to toggle source`
   Invokes the meth with the specified arguments, returning the method's return value.

```
m = 12.method("+")
```

```ruby
m.call(3)    #=> 15
m.call(20)   #=> 32
```

3. **arity → fixnum**

Returns an indication of the number of arguments accepted by a method. Returns a non-negative integer for methods that take a fixed number of arguments

```ruby
class C
  def two(a); end
  def three(*a);  end
  def four(a, b); end
  def five(a, b, *c);    end
  def six(a, b, *c, &d); end
end
=> :six


c = C.new
=> #<C:0x000000019556b8>

c.method(:two).arity     #=> 1
=> 1
c.method(:three).arity   #=> -1
=> -1
c.method(:four).arity    #=> 2
=> 2
c.method(:five).arity    #=> -3
=> -3
c.method(:six).arity     #=> -3
=> -3


"cat".method(:size).arity      #=> 0
"cat".method(:replace).arity   #=> 1
"cat".method(:squeeze).arity   #=> -1
"cat".method(:count).arity     #=> -1
```

4. **clone → new_method**

```ruby
class A
 def foo
```

```
   return "bar"
 end
end

m = A.new.method(:foo)
=> #<Method: A#foo>
m.call # => "bar"
n = m.clone.call # => "bar"

m.object_id != n.object_id
New method is created, not referencing to the same method.
```

5. **inspect → string**
Returns the name of the underlying method.

```
"cat".method(:size).inspect
=> "#<Method: String#size>"
```

6.**name → symbol**
Returns the name of the method.

```
"cat".method(:size).name
=> :size
```

7. **original_name → symbol**
Returns the original name of the method.

```
"cat".method(:size).original_name
=> :size
```

8. **owner → class_or_module**
Returns the class or module that defines the method.

```
"cat".method(:size).owner
=> String
```

9. **parameters → array**
Returns the parameter information of this method.

```
"cat".method(:size).parameters
=> []
```

10. receiver → object
Returns the bound receiver of the method object.

```
"cat".method(:size).receiver
=> "cat"
```

11. **Yielding to blocks**

```
def method(arg1,arg2)
 puts "First we are here:  #{arg1}"
 yield
 puts "Finally we are here:  #{arg2}"
 yield
end
method('start Method','end Method') { puts "Inside the yield" }

#> First we are here:  start Method
#> Inside the yield
#> Finally we are here:  end
#> Inside the yield

def simple(arg)
 puts "Before yield"
 yield(arg)
 puts "After yield"
end
simple('Nuts') { |name| puts "My name is #{Nuts}" }

#> Before yield
#> My name is Nuts
#> After yield
```

12. **Construction of Iteration using yield**
This is how iterators works:

```
def forEach(num)
 num.times do |i|
   yield(i)
```

```
 end
end

forEach(5) { |i| puts "number #{i}" }
number 0
number 1
number 2
number 3
number 4
```

In fact, it is with yield that things like foreach, each and times are generally implemented in classes

13. **block_given?**
If you want to find out if you have been given a block or not, use block_given?

```
class Person
 def names
   result = []
   @persons.each do |per|
     if block_given?
       yield(per.name)
     else
       result.push(per.name)
     end
   end

   result
 end
end
```

This example assumes that the Person class has an @persons list that can be iterated with each to get objects that have person names using the name method. If we are given a block, then we'll yield the name to the block, otherwise we just push it to an array that we return.

15. **Capturing undeclared keyword arguments (double splat)**
The ** operator works similarly to the * operator but it applies to keyword parameters.

```ruby
def method(required_key:, optional_key: nil, **other_options)
 other_options
end

method(required_key: 'Test', foo: 'Foo', bar: 'Bar')
#> { :foo => "Foo", :bar => "Bar" }
```

In the above example, if the **other_options is not used, an ArgumentError: unknown keyword: foo, bar error would be raised.

```ruby
def method(required_key:, optional_key: nil)
      puts "Testing...."
end

method(required_key: 'Test', foo: 'Foo', bar: 'Bar')
#> ArgumentError: unknown keywords: foo, bar
```

Global Methods

Instance Methods

Operator Methods

```ruby
def plus(other)
 # Define binary plus operator: x+y is x.+(y)
self.concat(other)
end

> "a".plus("n")
 => "an"
```