Ruby Metaprogramming

Metaprogramming

- Metaprogramming is writing code that manipulates language constructs at runtime.

Agenda

- Object, Class and Module
- Singleton Methods and Metaclass
- Block
- Eval family

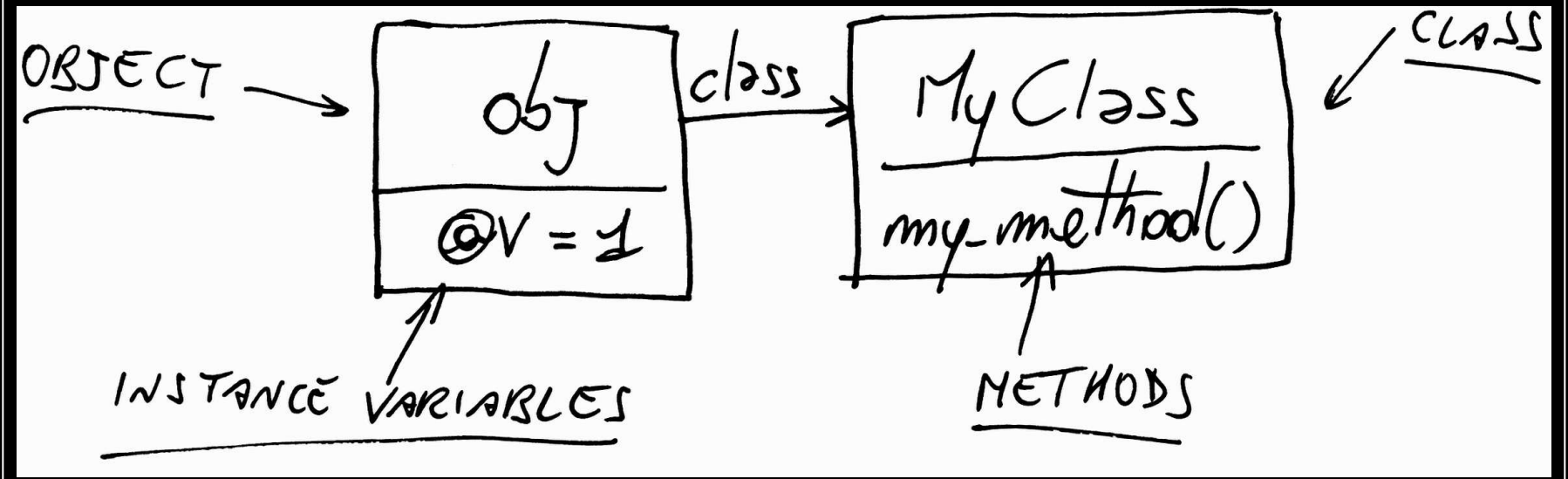
Part 1. Object, Class and Module

- Object introspection
- Classes
- Module
- Singleton Method
- Method Lookup
- Dynamic Methods
- Hook methods

Object Introspection

```
class MyClass
  def my_method
    @val = 1
  end
end

obj = MyClass.new
obj.class #=> MyClass
obj.class.superclass #=> Object
obj.my_method
obj.instance_variables #=> ["@val"]
obj.methods.grep(/my_/) #=> ["my_method"]
```

Objects of the same class share methods, but they don't share instance variables.



Metaprogramming Ruby p.38

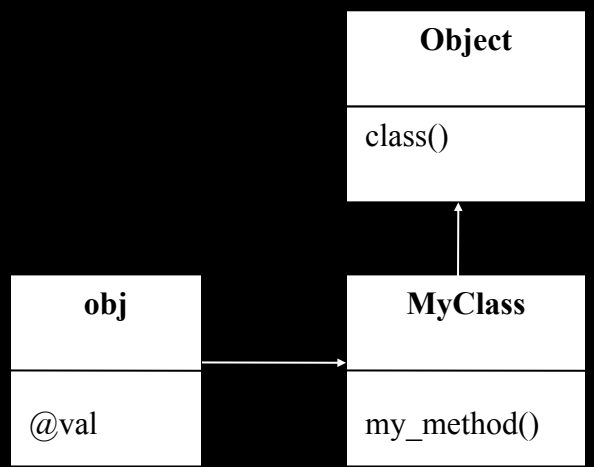
Classes

- Classes themselves are nothing but objects.
- Classes are instances of a class called Class.

```
MyClass.class #=> Class
MyClass.methods #=> ["inspect", ...]
MyClass.class.superclass #=> Object
MyClass.class.superclass.superclass #BasicObject
```

Method Lookup

- One step to the right, then up.



Open Class

- You can always reopen existing classes, and modify them.

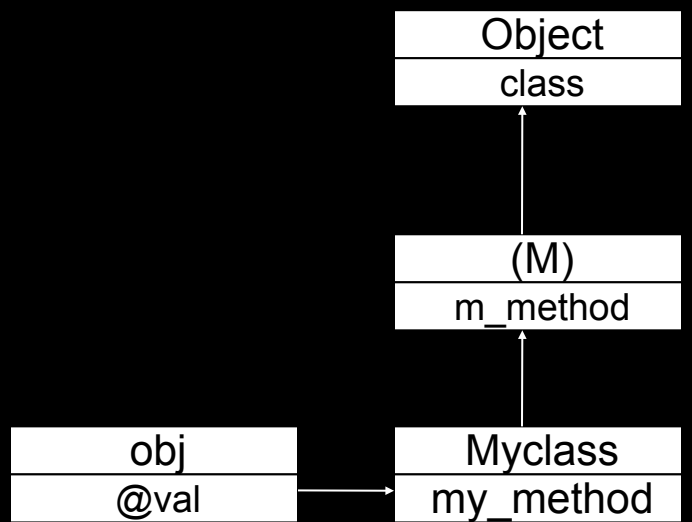
```
class Fixnum
  def hour
    self * 3600
  end

  def hours
    self * 3600
  end
end

1.hour #=> 3600
3.hour #=> 10800
```

Module

- Modules are also objects.
- When you include a module:
  - Ruby creates an anonymous class wraps the module.
  - Ruby then insert anonymous class into the ancestors chain.

Module

```ruby
module M
  def m_method
    puts "M method"
  end
end

class MyClass
  include M
end

obj.methods.grep(/m_/) #=> ["m_method"]
```



Singleton Method

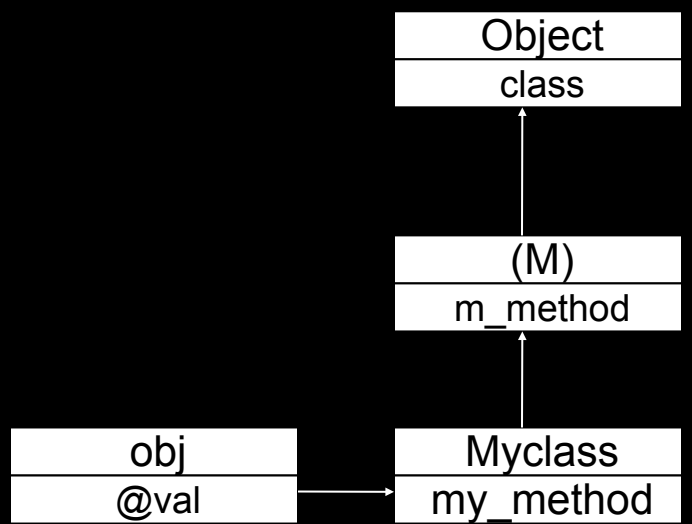
```
def obj.my_singleton_method
  puts "my singleton method"
end

obj.my_singleton_method #=> my singleton method
obj.singleton_methods #=> => ["my_singleton_method"]
```

Class Method are Singleton Method

```ruby
class MyClass
  def self.class_method_1
    puts "My class method 1"
  end
end

def MyClass.class_method_2
  puts "My class method 2"
end

MyClass.singleton_methods #=> ["class_method_2", "class_method_1"]
```

Metaclass

- Where is singleton method?
- An object can have its own special, hidden class named Metaclass.
- Also named eigenclass.



Metaclass

```
metaclass = class << obj
  self
end
metaclass.class #=> Class
metaclass.superclass #=> MyClass
metaclass.instance_methods.grep(/my/)
 #=> ["my_singleton_method", "my_method"]
```

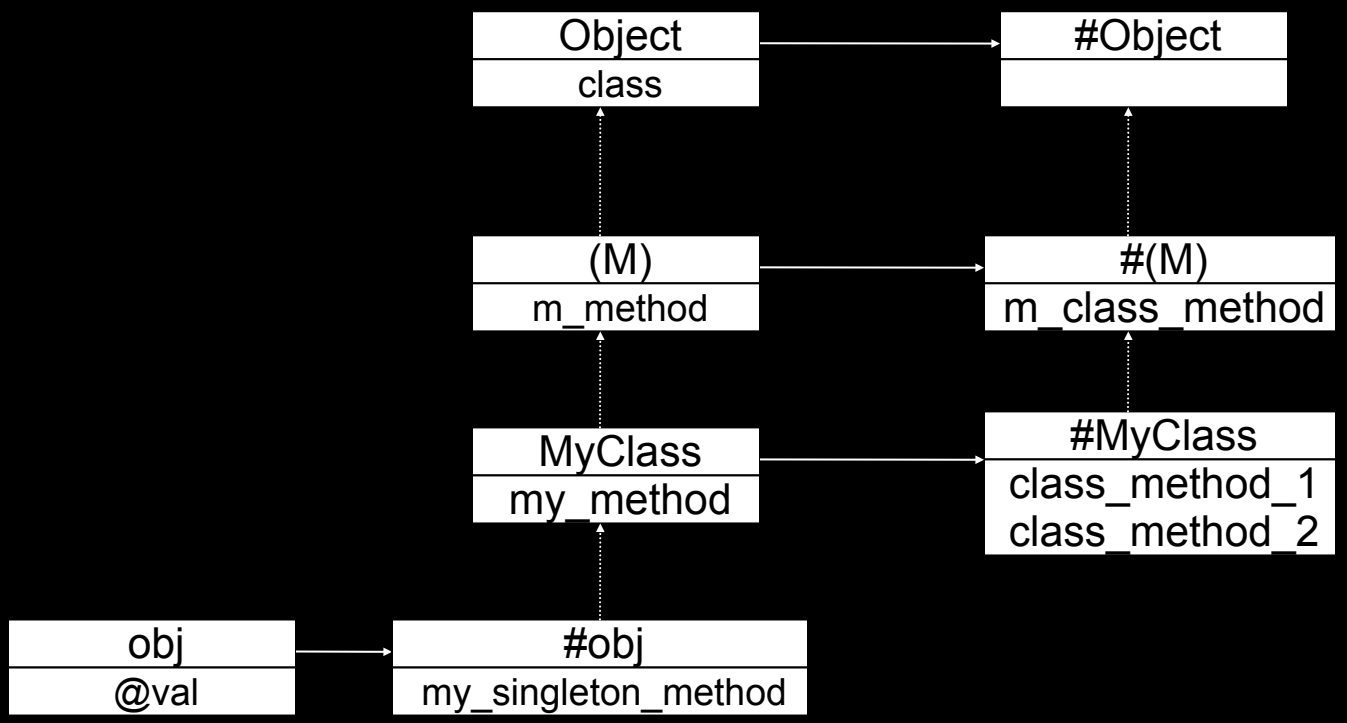
Define Class Methods

```
class MyClass
  class << self
    def class_method_1
      puts "My class method 1"
    end
  end
end

MyClass.class_method_1
```

What about module?

```
module M
  class << self
    def m_class_method
      puts "M class method"
    end
  end
end

class MyClass
  include M
end

MyClass.m_class_method #NoMethodError

class MyClass
  extend M
end

MyClass.m_class_method #=> M class method
```

Method Lookup Review



Define Methods Dynamically

```
# just a key/value pair
class SystemConfig < ActiveRecord::Base
  class << self

    [:column1, :column2, :column3].each do |name|
      define_method(name) do
        SystemConfig.find_by_key( name.to_s )
      end
    end

  end
end

SystemConfig.column1
SystemConfig.column2
```

Calling Method Dynamically

```ruby
class MyClass
  def my_method(arg)
    arg * 3
  end
end

obj = MyClass.new
obj.send(:my_method, 2) #=> 6
```

```ruby
[:column1, :column2, :column3].each do |name|
  SystemConfig.send(name)
end
```

method_missing()

- If method can't be find, then self calling method_missing().
- It's an instance method of Kernel that every object inherits.
- It responded by raising a NoMethodError.

Overriding method_missing()

```ruby
# from Ruby Metaprogramming p.77
class MyOpenStruct
  def initialize
    @attributes = {}
  end
  def method_missing(name, *args)
    attribute = name.to_s
    if attribute =~ /=$/
      @attributes[attribute.chop] = args[0]
    else
      @attributes[attribute]
    end
  end
end

icecream = MyOpenStruct.new
icecream.flavor = "vanilla"
icecream.flavor # => "vanilla"
```

Hook methods

- inherited()
- included()

```
module M2
  def self.included(base)
    puts "M2 included"
    base.extend(ClassMethods)
  end

  module ClassMethods
    def m4
      "Hello! m4 here"
    end
  end
end

class MyClass
  include M2
end

MyClass.m4
#=> M2 included
#=> Hello! m4 here
```

Part 2. Block

- Block
- Scope
- Callable Objects

Block

```ruby
class MyClass
  def block_method(a,b)
    return a + yield(a,b) if block_given?
    'no block'
  end
end

obj.block_method(1,2) #=> no block
obj.block_method(1,2) { |x, y| x * y }  #=>3
```

Scope

```ruby
def my_method
  x = "Goodbye"
  yield("weijen")
end

x = "Hello"
my_method { |y| "#{x}, #{y}" } #=> Hello, weijen
```

(But don’t do that)

Scope Gates

- Class definitions
- Module definitions
- methods

Proc Object

```
inc = Proc.new {|x| x + 1}
# Deffered Evaluation
inc.call(2) #=> 3

dec = lambda {|x| x - 1}
dec.class #=> #<Proc:0x018a7838@(irb):147>
dec.call(2) #=> 1
```

The & Operator

- A block can be an argument to a method.
- Block argument:
  - must be the last in the list of arguments.
  - Must prefixed by an & sign.
- & operator convert the block to a Proc object.
- & operator also can convert the Proc object to block.

The & Operator

```ruby
def math(a, b)
  yield(a, b)
end

def teach_math(a, b, &operation)
  puts "Let's do the math:"
  puts math(a, b, &operation)
end

teach_math(2, 3) {|x, y| x * y}

#=> Let's do the math:
#=> 6
```

Part 3. Eval* family

- eval
- instance_eval
- module_eval(class_eval)

Kernel#eval()

- it takes a string of code.
- It executes the code in the string.

```
str = "2 * 3"
eval(str)  #=> 6
```

The troble with eval()

- Your editor’s features mayn’t support.
- Not easy to read and modify.
- Not easy to debug.
- Code Injection

```
def explore_array(method)
  code = "['a', 'b', 'c'].#{method}"
  eval code
end

explore_array("reverse") #=> ["c", "b", "a"]

explore_array("object_id;Dir.glob(*)") #=> 很可怕，不要看
```

Object#instance_eval

```
obj #=> #<MyClass:0x18ca48c>

obj.instance_eval do
  self
end
#=> #<MyClass:0x18ca48c>
```

Module#class_eval(module_eval)

```
def add_method_to(clazz)
  clazz.class_eval do
    def m1
      "Hello! m1 here"
    end

    def self.m2
      "Hello! m2 here"
    end
  end
end

add_method_to(MyClass)
obj.m1 #=> "Hello! m1 here"
MyClass.m2 #=> "Hello! m2 here"
```

Classes themselves are nothing but objects.

```
MyClass.instance_eval do
  def m3
    "Hello! m3 here"
  end
end

MyClass.m3 #=> "Hello! m3 here"
```