Ruby Metaprogramming

Metaprogramming

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

Agenda

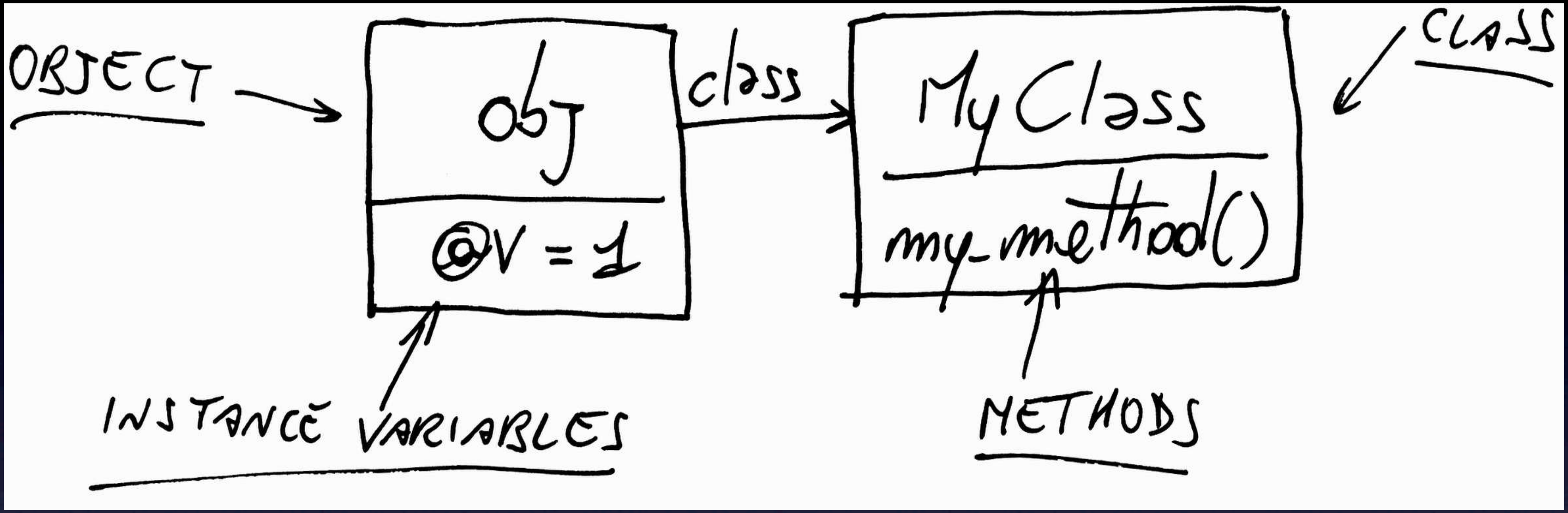
- Object, Class and Module
- Singleton Methods and Metaclass
- Eval Family

Object, Class and Module

Object Introspection

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

obj = MyClass.new
puts obj.class
puts obj.class.superclass
puts obj.my_method
puts obj.instance_variables
puts obj.methods.grep(/my_/)
```



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

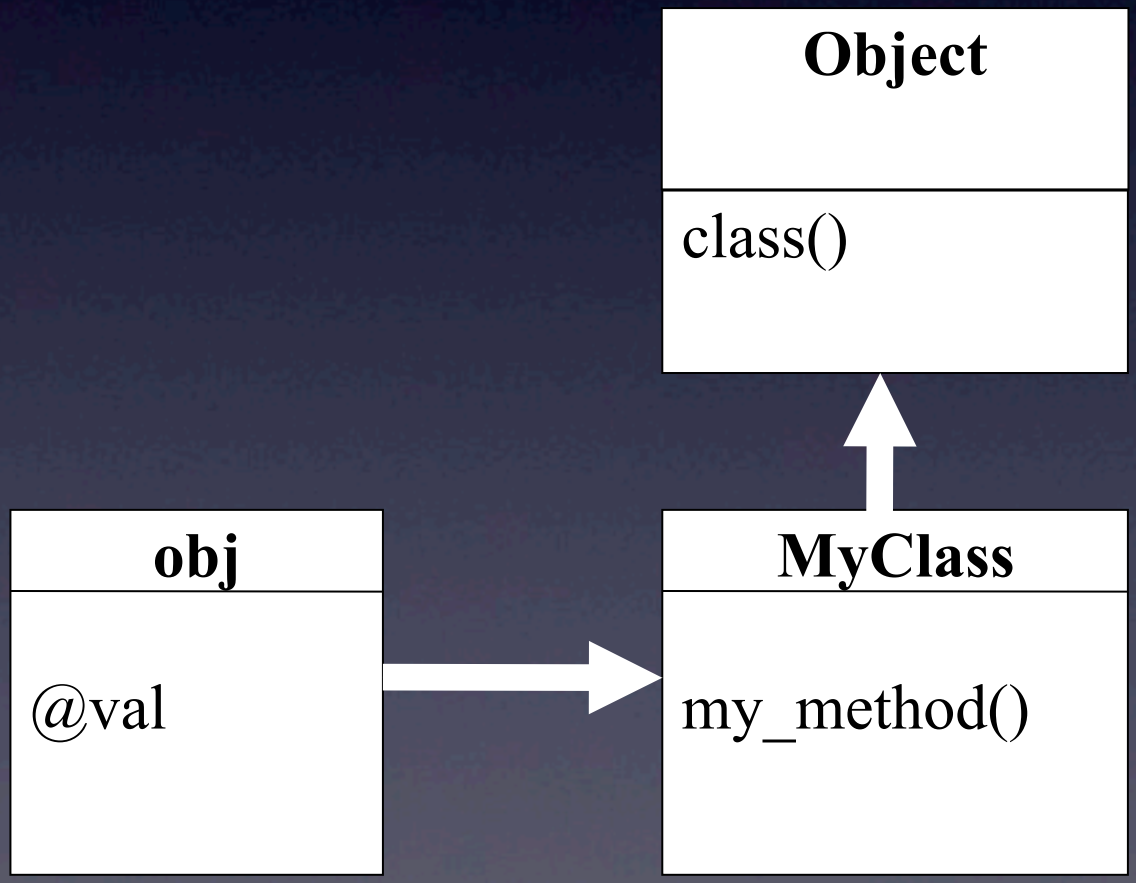
Classes

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

```
puts MyClass.class
#  => Class
puts MyClass.methods
puts MyClass.class.superclass
#  => Module
puts MyClass.class.superclass.superclass
#  => Object
```

Method Lookup

- One step to the right, then up.



Open Class

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

```ruby
class Fixnum
  def hour
    self * 3600
  end

  def hours
    self * 3600
  end
end
puts 1.hour
puts 3.hours
```

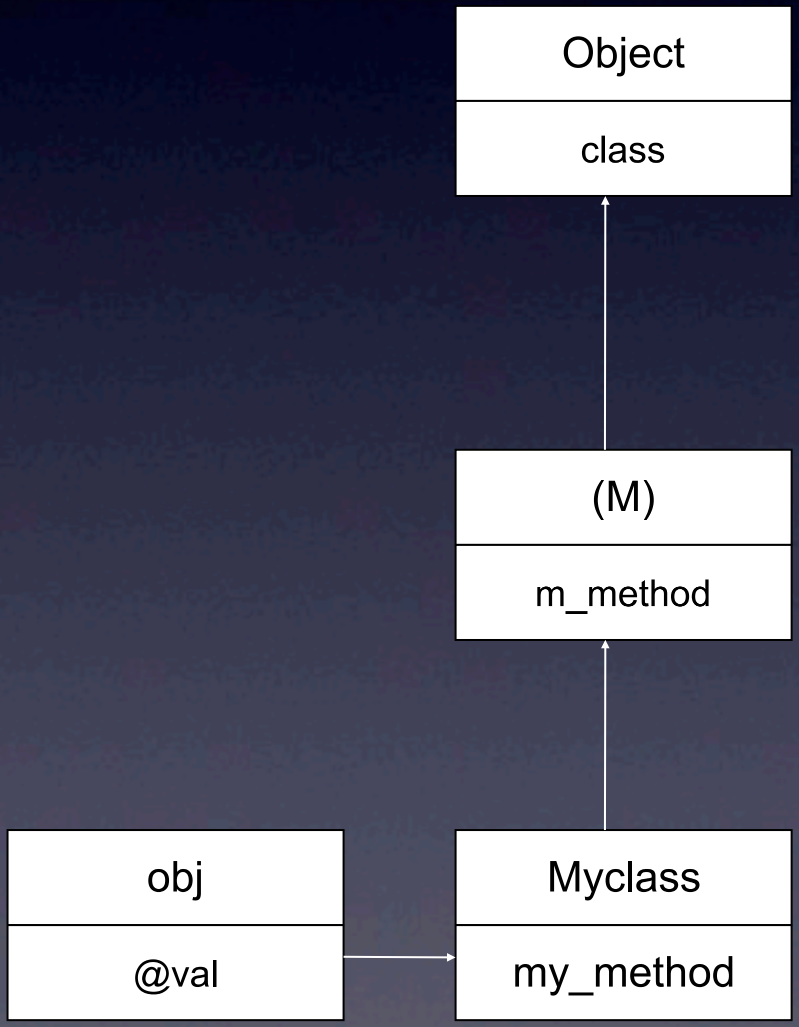
Module (I)

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

Module (II)

```
module M
  def m_method
    puts "M Method"
  end
end

class MyClass
  include M
end
obj = MyClass.new
obj.m_method
puts obj.methods.grep(/m_/)
```



Singleton Methods and Metaclass

Singleton Method

```
class MyClass
end

obj = MyClass.new
def obj.my_singleton_method
  puts "my singleton method"
end

obj.my_singleton_method
puts obj.singleton_methods
```

Class Method are Singleton Method

```ruby
class MyClass
  def self.class_method_1
    puts "class_method_1"
  end
end

def MyClass.class_method_2
  puts "class_method_2"
end

puts MyClass.singleton_methods
```

Metaclass (I)

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

Metaclass (II)

```
metaclass = class << obj
    self
end

puts metaclass.class
puts metaclass.superclass
puts metaclass.instance_methods.grep(/my_/)
```

Define Class Method

```ruby
class MyClass
  class << self
    def class_method_2
      puts "class_method_2"
    end
  end
end

MyClass.class_method_2
```

What about Module?

```
module M
  class << self
    def m_method_2
      puts "m_method_2"
    end
  end
end
class MyClass
  include M
  extend M
end
#MyClass.m_method_2
#MyClass.new.m_method_2
M.m_method_2
```

Define Methods Dynamically

```
class SystemConfig
  class << self
    %w(column1 column2 column3).each do |name|
      define_method(name) do
        puts "Hi #{name}"
      end
    end
  end
end

SystemConfig.column1
```

Calling Methods Dynamically

```
%w(column1 column2 column3).each do |name|
  SystemConfig.send(name)
end
```

method_missing()

- If method can’t be found, then self calling method_missing().
- method_missing is an instance method of Kernal that every object inherits.
- It responded by raising a NoMethodError.

Overriding method_missing()

以Rails的ActiveModel 作例子：

https://github.com/rails/rails/blob/master/activemodel/lib/active_model/attribute_methods.rb

Hook Methods (I)

```ruby
module M
  def self.included(base)
    puts "M was included"
    base.extend(ClassMethods)
  end

  module ClassMethods
    def m_method
      puts "m_method"
    end
  end
end
```

Hook Methods (II)

- inherited
- included
- extended

Eval Family

Kernel#eval()

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

```
str = "2 * 3"
puts eval(str)
```

The Troubles with eval

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

Object#instance_eval

```
class MyClass
end
obj = MyClass.new
puts obj
obj1 = obj.instance_eval do
  self
end
puts obj1
```

Module#class_eval

```ruby
def add_mothod_to(clazz)
  clazz.class_eval do
    def m1
      puts "m1"
    end

    def self.m2
      puts "m2"
    end
  end
end
```

Classes are objects

```ruby
class MyClass
end

MyClass.instance_eval do
  def m3
    puts "m3"
  end
end

MyClass.m3
```