Ruby Meta-Programming

Sang Shin
JPassion.com
"Code with Passion!"

Topics

- What is and Why Meta-programming?
- Ruby language characteristics (that make it a great meta-programming language)
- Object#respond_to?
- Object#send
- Dynamic typing (and Duck typing)
- missing_method
- define_method

What is Meta-Programming?

What is Meta-Programming?

- Meta-programming is the writing of computer programs that write or manipulate other programs (or even themselves) as their data

Why Meta-Programming?

- Provides higher-level abstraction of logic
  - Easier to write code
  - Easier to read code
- Meta-programming feature of Ruby language is what makes Rails a killer application
  - For example, the dynamic finders in Rails such as "find_by_name", "find_by_name_and_hobby" are possible because of the Meta-programming feature of Ruby language

Ruby Language Characteristics that Make It a Great Meta-Programming Language

Ruby Language Characteristics

- Classes are open
- Class definitions are executable code
- Every method call has a receiver
- Classes themselves are objects

source: http://www.infoq.com/presentations/metaprogramming-ruby

Classes Are Open

- Unlike Java and C++, in Ruby, during runtime, methods and variables can be added to a class (including built-in core classes provided by Ruby such as *String* and *Fixnum*)

```
# define a new method called encrypt for String class
class String
  def encrypt
    tr "a-z","b-za"
  end
end

puts "cat"
puts "cat".encrypt
```

Classes Are Open

- Benefits
  - Applications can be written in higher level abstraction
  - More readable code
  - Less coding
- How it is used in Rails
  - Anyone can open up Rails classes and add new features (mostly methods) to them to suit their needs

Class Definitions are Executable Code

- Class definition is basically creating a new Class object during runtime
  - "class" is actually a method of Class class

```
# The log(msg) method is defined differently during runtime
class Logger
  if ENV['DEBUG']
    def log(msg)
      STDERR.puts "LOG: " + msg
    end
  else
    def log(msg)
    end
  end
end
```

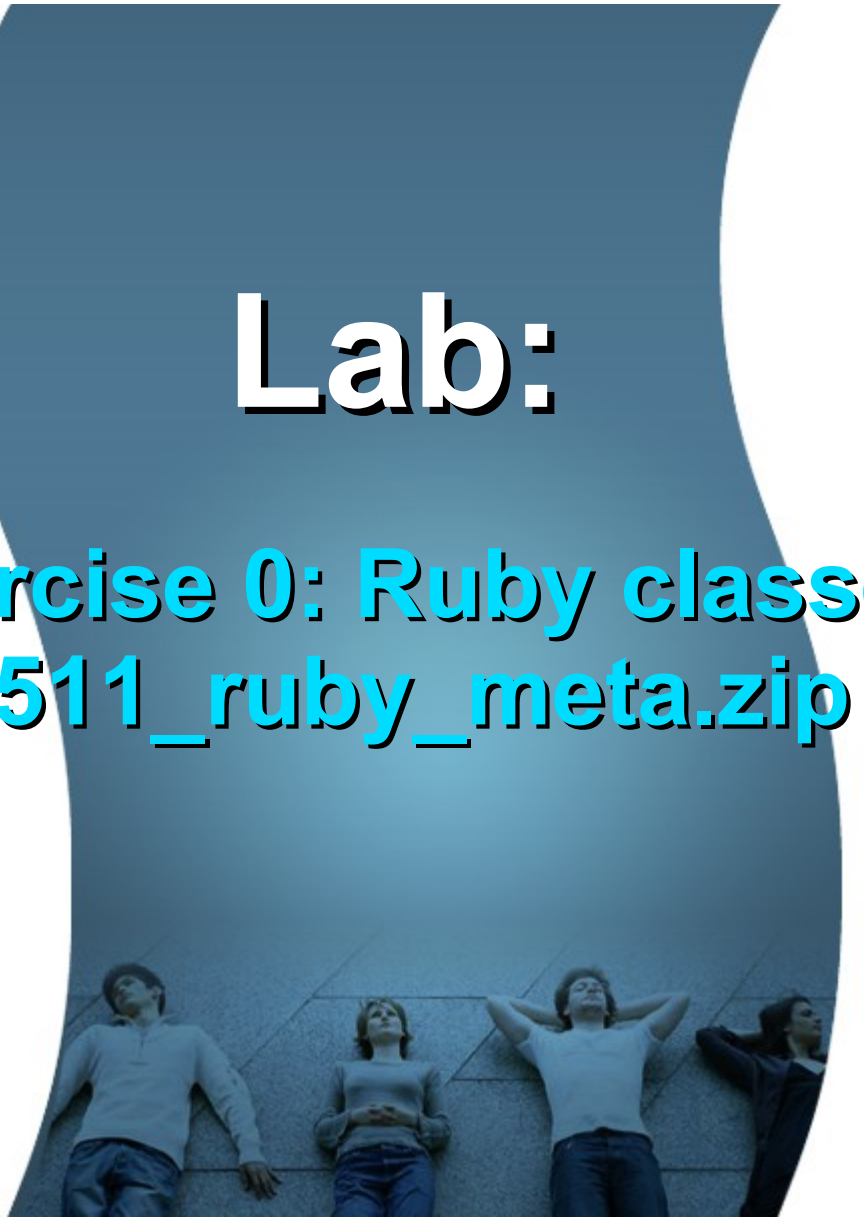
Classes Are Objects

- *String* class is an instance of *Class* class in the same way *Fixnum* class (or *Person* class) is an instance of *Class* class

```
class Person

  puts self          # Person
  puts self.class    # Class

  def self.my_class_method
    puts "This is my own class method"
  end

end
```

Lab:

Exercise 0: Ruby classes
5511_ruby_meta.zip



Object#respond_to?

What is Introspection?

- Being able to find information on an object during runtime
- Examples
  - Object#class
  - Object#methods
  - Object#class.superclass
  - Object#class.ancestors
  - Object#private_methods
  - Object#public_methods
  - ...

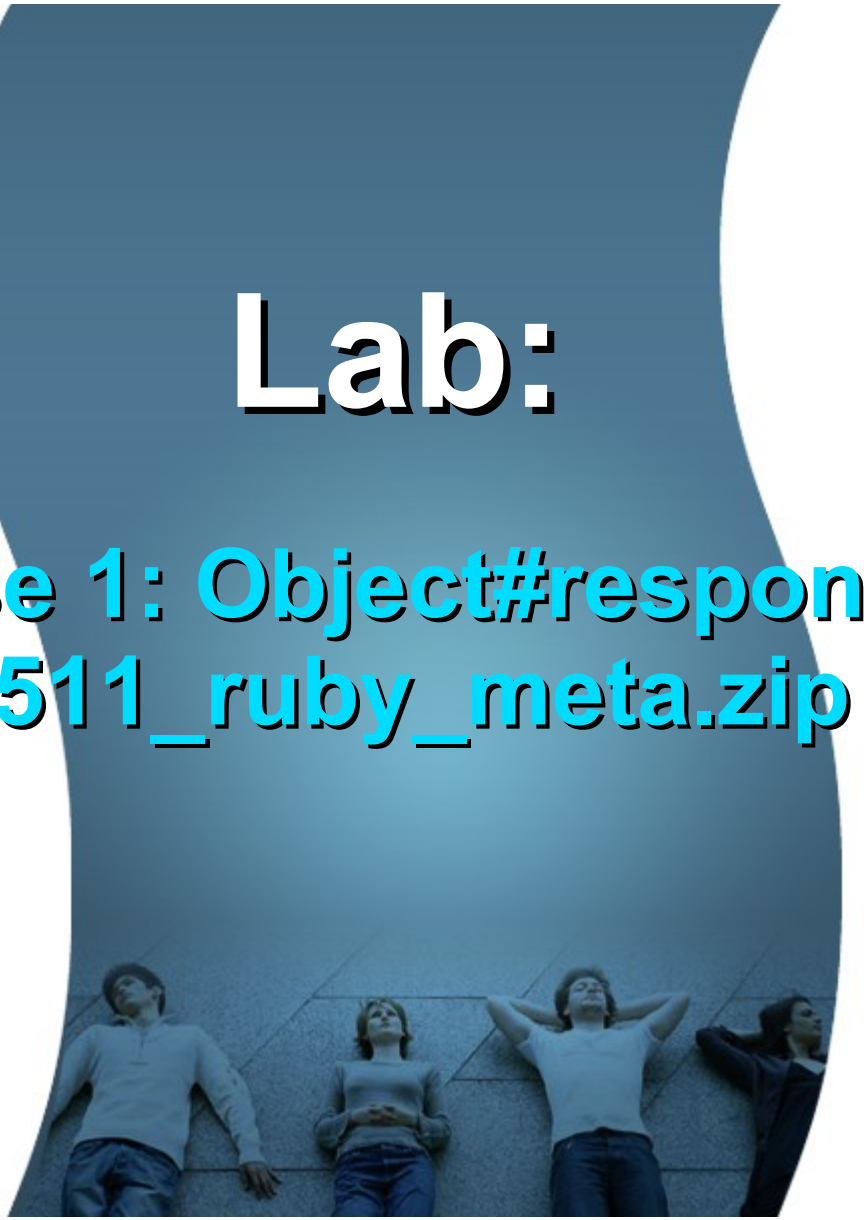
respond_to? method

- A method in the “Class” object
- Returns "true" if obj responds to the given method, in other words, if the class of the object has the given method

```
class Hello
  def myhellomethod(name)
  end
end

hello_instance = Hello.new
puts hello_instance.respond_to?(:myhellomethod)  # true
```

Lab:

Exercise 1: Object#respond_to?
5511_ruby_meta.zip



Object#send Method Class

Method Invocation in Ruby

- Calling a method directly by name is allowed as we know of
  - an_object_instance.hello("Good morning!")
- It is also possible to call a method through *send(..)* passing string, symbol, or variable as the name of the method
  - an_object_instance.send("my_method", args)
  - an_object_instance.send(:my_method, args)
  - amethod = :my_method
  - an_object_instance.send("#{amethod}", args)
- This allows calling different method during runtime depending on business logic, time of the day, etc
  - Example: I want to call "handle_good_customer()" for a good customer and "handle_bad_customer() for a bad customer

Example: obj.send(symbol [, args...])

- Invokes the method identified by symbol (or string), passing it any arguments specified.

```
class Klass
  def hello(*args)
    "Hello " + args.join(' ')
  end
end

k = Klass.new

# The following statements are equivalent
puts k.hello("gentle", "readers")          #=> "Hello gentle readers"
puts k.hello "gentle", "readers"           #=> "Hello gentle readers"
puts k.send("hello", "gentle", "readers")  #=> "Hello gentle readers"
puts k.send "hello", "gentle", "readers"   #=> "Hello gentle readers"
puts k.send(:hello, "gentle", "readers")   #=> "Hello gentle readers"
puts k.send :hello, "gentle", "readers"    #=> "Hello gentle readers"
```

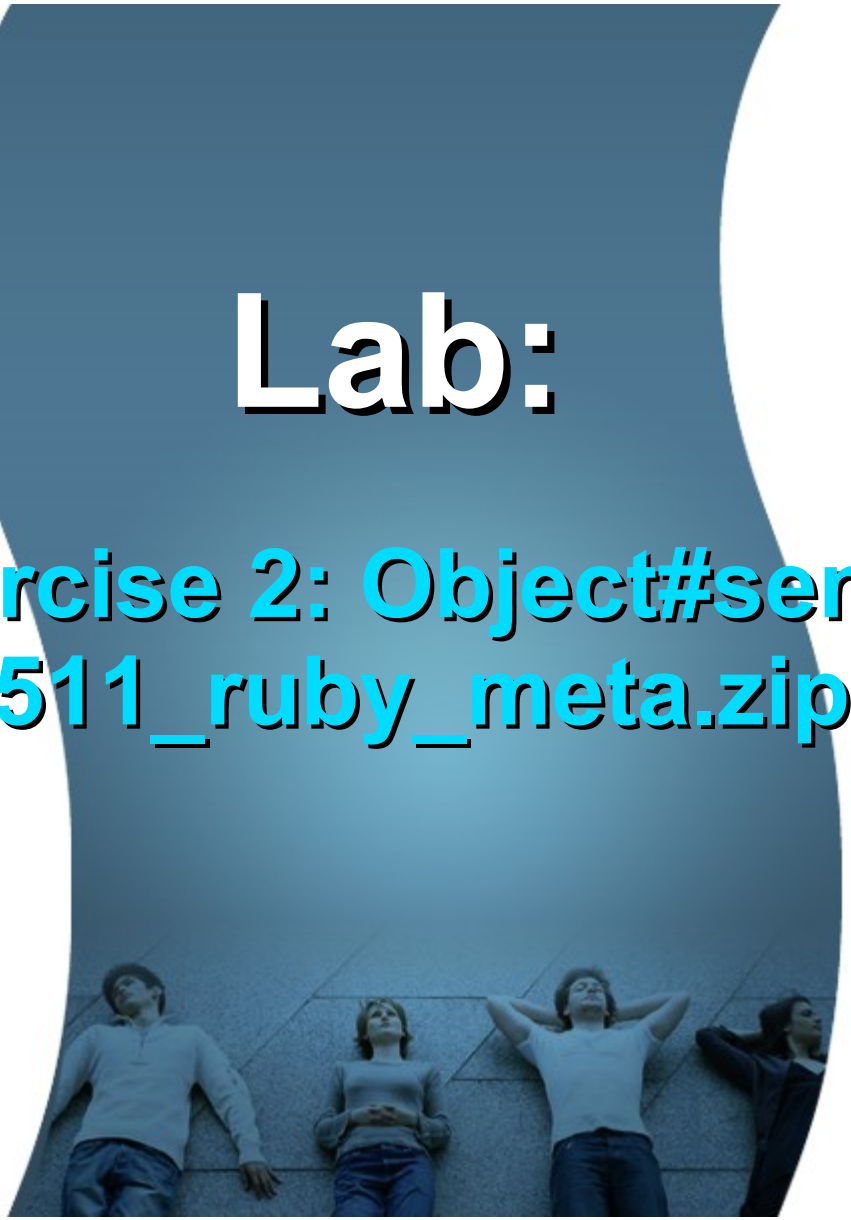
Method Class

- Method object represents a method
- You can invoke the method by invoking “call” method of the Method object

```
length_method = "Random text".method(:length)
p length_method.class        # Method
p length_method.call         # 11
```

Lab:

Exercise 2: Object#send
5511_ruby_meta.zip



Dynamic Typing (and Duck Typing)

What is Dynamic Typing?

- A programming language is said to use dynamic typing when type checking is performed at run-time (also known as "late-binding") as opposed to compile-time
- Examples of languages that use dynamic typing include
  - Ruby, PHP, Lisp, Perl, Python, and Smalltalk

What is Duck Typing?

- Duck typing is a style of dynamic typing in which an object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class
- The name of the concept refers to the duck test, attributed to James Whitcomb Riley, which may be phrased as “If it walks like a duck and quacks like a duck, I would call it a duck”.

Duck Typing Example (page 1)

```
# The Duck class
class Duck
  def quack
    puts "Duck is quacking!"
  end
end

# The Mallard class
class Mallard
  def quack
    puts "Mallard is quacking!"
  end
end
```

Duck Typing Example (page 2)

```
# If it quacks like a duck, it must be duck
def quack_em(ducks)
  ducks.each do |duck|
    if duck.respond_to? :quack
      duck.quack
    end
  end
end

birds = [Duck.new, Mallard.new, Object.new]

puts "----Call quack method for each item of the birds array. Only Duck and Mallard
     should be quacking."
quack_em(birds)
```

Lab:

Exercise 3: Dynamic Typing (Duck Typing)
5511_ruby_meta.zip

missing_method

NoMethodError Exception

- If a method that is not existent in a class is invoked, *NoMethodError* exception will be generated

```
# Let's say we defined Dummy class
class Dummy
end

puts "----Call a method that does not exist in the Dummy class
    and expect NoMethodError exception."
dummy = Dummy.new
dummy.call_a_method_that_does_not_exist
```

method_missing Method

- If *method_missing(m, *args)* method is defined in a class, however, it will be called (instead of *NoMethodError* exception being generated) when a method that does not exist is invoked

```
class Dummy
  def method_missing(a_method, *args)
    puts "There's no method called #{a_method} here -- so
    method_missing method is called."
    puts "   with arguments #{args}"
  end
end

dummy = Dummy.new
dummy.a_method_that_does_not_exist
```

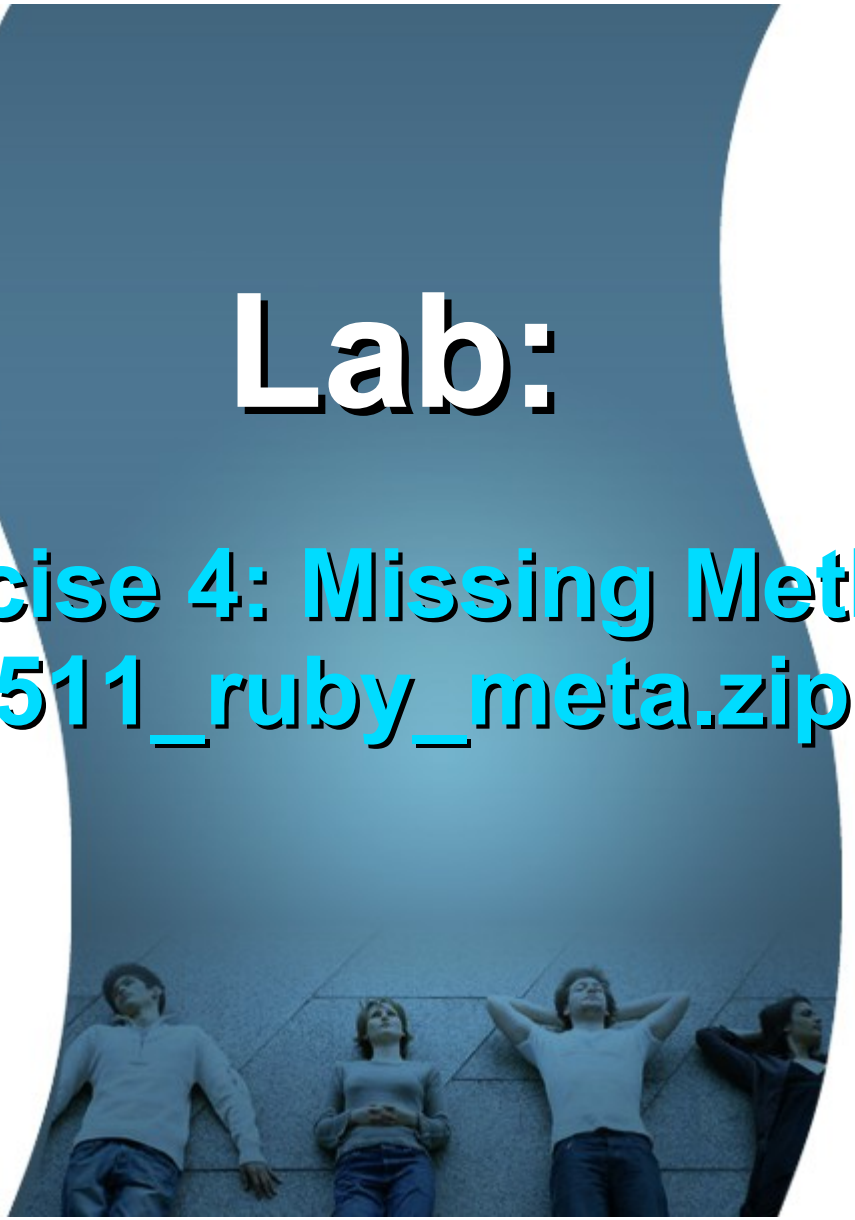
How method_missing Method is used in Rails

- Rails' *find_by_xxxx()* finder method is implemented through *method_missing*.

```
class Finder
  def find(name)
   # Rails (actually ActiveRecord) constructs a find() method with correct
   # set of  parameters
    puts "find(#{name}) is called"
  end

  def method_missing(name, *args)
     # code to handle the finder logic
  end
end

f = Finder.new
f.find("Something")
f.find_by_last_name("Shin")
f.find_by_title("Technology Architect")
```

Lab:

Exercise 4: Missing Method 5511_ruby_meta.zip



define_method

define_method

- The define_method defines an instance method in the receiver.

  define_method(symbol, method)
  define_method(symbol) { block }

- The *method* parameter can be a *Proc* or *Method* object
  - If a block is specified, it is used as the method body

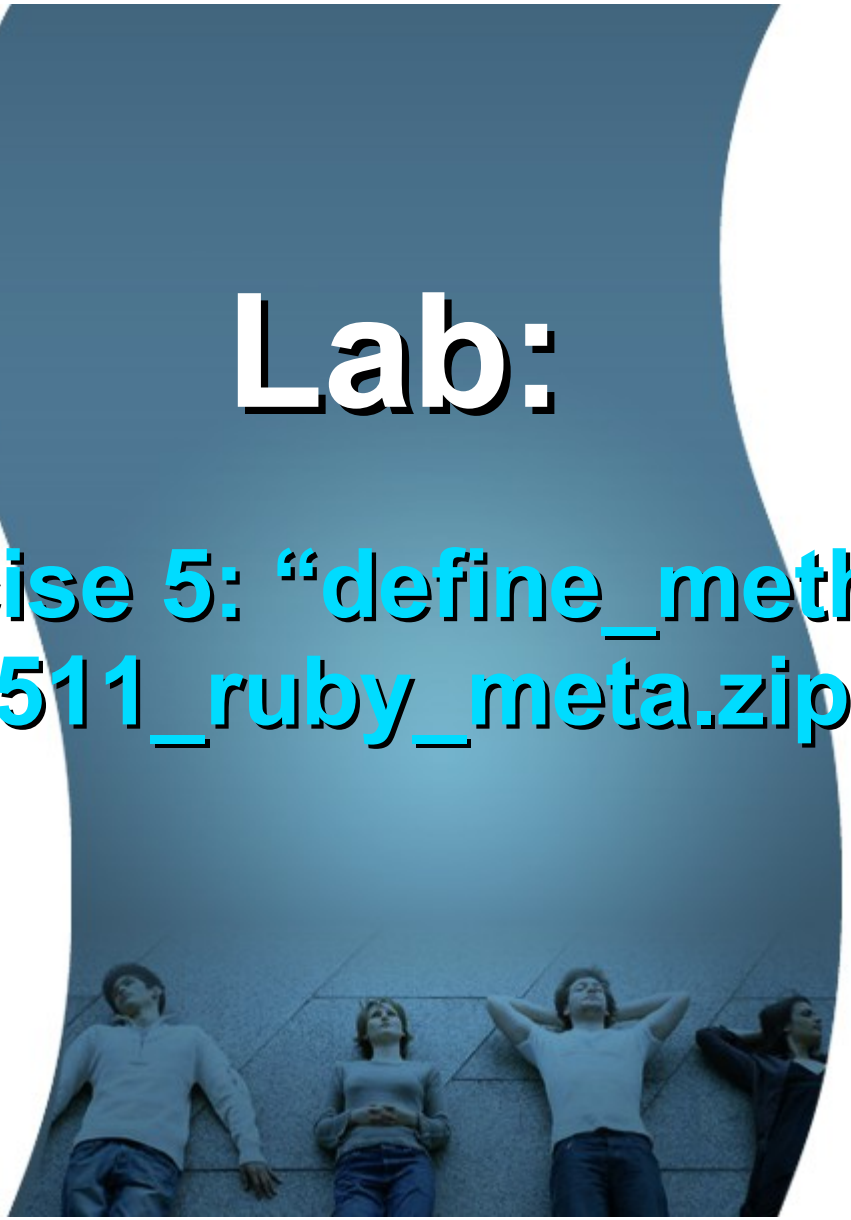
define_method

- An example of *define_method(symbol) { block }*

```
class Love
  define_method(:my_hello) do |arg1, arg2|
    puts "#{arg1} loves #{arg2}"
  end
end

love = Love.new
# my_hello is a method to call
love.my_hello("Barbara", "John")
```

Lab:

Exercise 5: “define_method”
5511_ruby_meta.zip



Code with Passion!
JPassion.com