Objects in Ruby

Sang Shin
Michèle Garoche
www.javapassion.com
"Learning is fun!"

Topics

- Objects in Ruby language
- Class object
- Dynamic method invocation through Object#send
- Self
- Singleton method

You have seen some slides in this presentation in other presentations. They are repeated here for completeness of this topic.

Objects in Ruby Language

In Ruby, Everything is an Object

- Like Smalltalk, Ruby is a pure object-oriented language — everything is an object.
- In contrast, languages such as C++ and Java are hybrid languages that divide the world between objects and primitive types.
- The hybrid approach results in better performance for some applications, but the pure object-oriented approach is more consistent and simpler to use.

What is an Object (in general sense)?

- Using Smalltalk terminology, an object can do exactly three things.
  - Hold state, including references to other objects.
  - Receive a message, from both itself and other objects.
  - In the course of processing a message, send messages, both to itself and to other objects.
- If you don't come from Smalltalk background, it might make more sense to rephrase these rules as follows:
  - An object can contain data, including references to other objects.
  - An object can contain methods, which are functions that have special access to the object's data.
  - An object's methods can call/run other methods/functions.

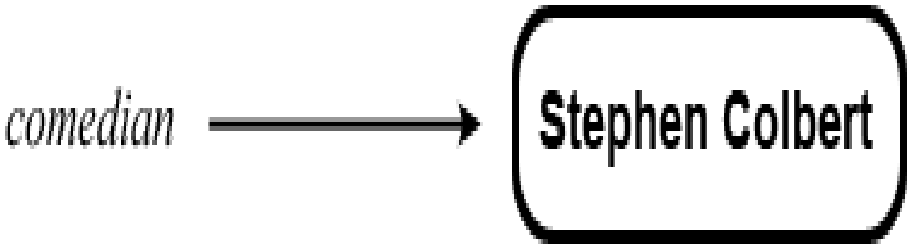
In Ruby, Everything Is An Object

- 'Primitives' are objects
  - *-1.abs* # call '*abs*' method of '*-1*' as an object
- nil is an object
  - *nil.methods* # get '*all*' methods of '*nil*' object
- Classes are objects
  - *Song.new* # invoke the *new* class method on 'Song' class
  - Create instances of themselves
- Code blocks are objects
  - They can be pass around, even as parameters
  - Also known as closure

Variables and Objects (in Ruby)

- Create a *String* object containing the text "*Stephen Colbert*". We also told Ruby to use the variable *comedian* to refer to this object. (Works the same as in Java)

```
>> comedian = "Stephen Colbert"
=> "Stephen Colbert"
```



Class Object

Class Definitions (in Ruby) are actually Executable Code

- Class definition is basically creating a new Class object during runtime
- Example: The *log(msg)* method in *Logger* class 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 is an instance of Class

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

end

# We will talk about self later on in this presentation
```

Dynamic Method Invocation through Object#send

Dynamic Method Invocation in Ruby

- In Ruby, an object's methods are not fixed at any compilation time but can be dynamically extended or modified at any point
- Calling a method directly by name is allowed as expected (like in Java)
  - *an_object_instance.hello("Good morning!")*
- It is also possible to invoke generically any object method by using a string or symbol variable to specify the target method
  - *an_object_instance.send("#{name_of_method}", args)*
  - *an_object_instance.send(:my_method, args)*

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

- Invokes the method identified by symbol, 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"
```

Self

Every Method Call Has a Receiver

- Default receiver is *self*
- *Self* represents the current object

Self in an Instance Method

```
class MyClass
  def my_instance_method

    puts "---- What is the object instance invoking (or receiving) this
     instance method?"
    puts self                       # <MyClass:0x1758500>

    puts "---- What is the class of this object instance?"
    puts self.class                 # MyClass

    puts "---- What is the parent class of the class of this object instance?"
    puts self.class.superclass      # Object

    puts "---- What is the class of the Object class?"
    puts self.class.superclass.class  # Class

  end
end

my_class = MyClass.new
my_class.my_instance_method    #<MyClass:0xae3364>
```

Self in an Class Method

```
class MyClass

  def self.my_class_method
    puts "---- What is the object instance receiving this class method?"
    puts self             # MyClass

    puts "---- Verify that MyClass class is an object instance of Class class."
    puts self.class       # Class

    puts "---- What is the parent class of MyClass class?"
    puts self.superclass  # Object

    puts "---- What is the parent class of Class class?"
    puts self.class.superclass  # Module

    puts "---- What is the parent class of Module class?"
    puts self.class.superclass.superclass  # Object
  end
end

MyClass.my_class_method
```

Self in an Top-Level Context

```
puts "---- What is the object instance receiving this puts method?"
puts self              # main

puts "---- What is the class of this object instance?"
puts self.class      # Object

# Top level methods are 'private instance methods of the Object class.
# Because top-level methods are private, you cannot call the with an
# explicit receiver; you can only call them with the default receiver,
# self.
def my_top_level_instance_method
  puts "---- What is the object receiving this instance method?"
  puts self  # main
end

my_top_level_instance_method

def self.my_top_level_class_method
  puts "---- What is the object receiving this class method?"
  puts self  # main
end

my_top_level_class_method
```

Singleton Method

What is Singleton Method?

- A method given only to a single object (not to all objects from a class) is called a singleton method.

Example: Singleton Method

```
# From http://www.rubyist.net/~slagell/ruby/singletonmethods.html

# In this example, test1 and test2 belong to same class, but
# test2 has been given a redefined size method and so they behave
# differently. A method given only to a single object is called
# a singleton method.

class SingletonTest
  def size
    25
  end
end

test1 = SingletonTest.new
test2 = SingletonTest.new

def test2.size
  10
end

puts "----test1.size = #{test1.size}"  # 25
puts "----test2.size = #{test2.size}"  # 10
```

Example2: Singleton Method

```
class C
end

foo = C.new

def foo.bar
  puts "from foo.bar"
end

# This is another way a singleton method can be defined
class << foo
  def bar2
    puts "from foo.bar2"
  end
end

foo.bar        #<Object:0x15356d5>
foo.bar2        #<Object:0x15356d5>

foo2 = C.new
#foo2.bar       # Compile error - undefined method - expected
#foo2.bar2       # Compile error - undefined method - expected
```

Thank you!

JavaPassion.com provides "Satisfaction Guaranteed" Instructor-led codecamps.

Please check upcoming codecamps from http://www.javapassion.com/codecamps