Crash Course in Ruby Metaprogramming

JavaZone Lightning Talk

Who? Sindre Moen

When? September 12, 2012

Overview of this talk

- The Ruby Object
- Singleton Classes
- Hook Methods
- Defining and Calling Methods
- evals
- When (not) to Metaprogram

The Ruby Object

- Everything is an object

```ruby
Object.class # => Class
42.class     # => Fixnum
false.class  # => FalseClass
nil.class    # => NilClass
```

- Objects and classes are open

```ruby
def Fixnum.meaning_of_life_the_universe_and_everything?
  self == 42
end

20.meaning_of_life_the_universe_and_everything? # => false
42.meaning_of_life_the_universe_and_everything? # => true
```

Singleton Classes

- Nothing to do with the *singleton pattern*
- *Anonymous* class between an object and its class

```
leet_array = []
def leet_array.size () 1337 end

leet_array.size                # => 1337
leet_array.singleton_class     # => #<Class:#<Array:0x9048fcc>>
leet_array.singleton_methods # => [:size]
```

- Used to define “class methods”

```
class Foo
  def self.bar () "bar" end
end
```

Hook Methods

method_missing

```ruby
class Hash
  def method_missing(method, *params)
    method = method.to_sym

    if self.keys.collect(&:to_sym).include? method
      return self[method]
    end

    super
  end
end

{foo: 'bar' }.foo # => bar
```

const_missing

```ruby
class Module
  orig_const_missing = instance_method :const_missing

  define_method :const_missing do |name|
    if name.match /^U([0-9a-zA-Z]{4})$/
      [$1.to_i(16)].pack("U*")
    else
      orig_const_missing.bind(self).call(name)
    end
  end
end

U0041 # => "A"
```

Defining and Calling Methods

- Ruby has two ways of defining methods

```ruby
def foo
  "bar"
end

define_method :baz do
  "qux"
end
```

- `define_method` can handle dynamic method names

```ruby
class Request

  { accept!:  :accepted,
    decline!: :declined,
    await!:   :pending
  }.each do |name, status|
    define_method name do
      @status = status
    end
  end

end

Request.new.accept!  # => :accepted
Request.new.decline! # => :declined
Request.new.await!   # => :pending
```

Defining and Calling Methods

- Two ways to call methods

```
'foo'.upcase       # => "FOO"

'foo'.send :upcase # => "FOO"
```

- `send` can handle dynamic method names

```
class Object
  def try(*args, &block)
    return nil if nil?
    send(*args, &block)
  end
end

@duck = Duck.new; @duck.try :speak # => "Quack!"
@duck = nil;      @duck.try :speak # => nil (does not raise an error)
```

evals

- Three types of `eval` methods
  - `class_eval`
  - `instance_eval`
  - `eval`

evals – `class_eval`

- Defined on `Module`
- Receiver is a class or a module

```
Fixnum.class_eval do
  def is_this_it?
    self == 42
  end
end

42.is_this_it? # => true
```

- Receiver is the `Person` *class*
- Hence, the method is defined on *objects* of Fixnum

evals – `instance_eval`

- Defined on `Object`
- Receiver is an object

```
Fixnum.instance_eval do
  def the_meaning_of_life_the_universe_and_everything
    42
  end
end

Fixnum.the_meaning_of_life_the_universe_and_everything # => 42
```

- Receiver is the `Fixnum` *object*
- Hence, the method is defined on the Fixnum *class*

evals – `eval`

- "`eval` is evil"
- Evaluates a string as Ruby code

```
eval 'puts "foo bar baz qux"'
```

- Very powerful, but also dangerous
- Susceptible of malicious code injection
- You probably don't need it . . . ever

When (not) to Metaprogram?

Don't do it ...

- if "simply programming" is enough
- just because it requires less code

Do it ...

- when it makes your code nicer and DRYer
- when it makes your code easier to maintain
- as simply as possible