Ruby Metaprogamming Cheat Sheet (By Example)

Introspection

Listing Methods

List the `String` type's *class* methods, sorted:

```
‣ String.methods.sort
    ["<", "<=", "<=>", ..., "to_a", "to_s", ...]
```

List the `String` type's *public/private class* methods whose name contains the word "method":

```
‣ String.public_methods.grep(/method/)
    ["private_class_method", "method", ...]
‣ String.private_methods.grep(/method/)
    ["singleton_method_added", ...]
```

Three different ways to list the `String` type's *instance* methods that start with "to":

```
‣ String.instance_methods.grep(/^to/)
‣ String.new.methods.grep(/^to/)
‣ “string”.methods.grep(/^to/)
    ["to_str", "to_i", "to_f", "to_a", "to_s", ...]
```

List the `String` type's *public instance* methods, but not the *inherited* methods[1]:

```
‣ (String.instance_methods -
  Object.instance_methods).sort
    ["%", "*", "+", "<", "<<", "<=", "<=>", ...]
```

List the `String` type's *public/protected/private instance* methods, but not the inherited methods:

```
‣ (“string”.public_methods -
  Object.public_instance_methods).sort
    ["%", "*", "+", "<", "<<", "<=", "<=>", ...]
‣ (“string”.protected_methods - Object.
  protected_instance_methods).sort
    []
‣ (“string”.private_methods -
  Object.private_instance_methods).sort
    []
```

Finding Methods

Get the `String` type's *class* method named `ancestors`:

```
‣ String.method(:ancestors)
    #<Method: Class(Module)#ancestors>
```

Get the `String` type's *instance* method named `gsub`:

```
‣ “a string”.method(:gsub)
    #<Method: String#gsub>
```

Getting and Setting Attribute Values

Get and set a *class* attribute `cattr` in class `Foo`.

```
‣ class Foo
‣   @@cattr = “cattr”
‣ end
```

[1] Actually, the list includes `Comparable` and `Enumerable` methods, modules which `String` includes.

```
‣ Foo.class_eval do
‣   v = class_variable_get(:@@cattr)
‣   class_variable_set(:@@cattr, “foo”)
‣ end
    “foo”
```

Get and set an *instance* attribute `attr` in class `Foo`.

```
‣ class Foo
‣   def initialize
‣      @attr = “attr”
‣   end
‣ end
‣ Foo.new.instance_eval do
‣   v = instance_variable_get(:@attr)
‣   instance_variable_set(:@attr, “bar”)
‣ end
    “bar”
```

Finding Types

Get the list of *top-level classes*:

```
‣ Class.constants.find_all do |x|
‣   Class.const_get(x).class==Class
‣ end
    ["TrueClass", "SecurityError", "Array", ...]
```

Print all *classes* in the current runtime:

```
‣ ObjectSpace.each_object(Class) {|c| p c}
    IRB::Context ...
```

Print all *modules* in the current runtime:

```
‣ ObjectSpace.each_object(Module) {|c| p c}
    IRB::Context ...
```

Get the `String` type’s parent types (classes and modules):

```
‣ String.method(:ancestors)
    #<Method: Class(Module)#ancestors>
```

Finding Objects

Print all *instances* of type `Integer` in the current runtime:

```
‣ ObjectSpace.each_object(Integer) {|i| p i}
    9223372036854775807 ...
```

Manipulating “Stuff”

Introducing New Elements

Include the `Enumerable` module in type `ThreeIntegers`:

```
‣ class ThreeIntegers
‣   include Enumerable
‣   def each; ...; end
‣ end
‣ ThreeIntegers.new.each {|i| i*2}
    [0, 2, 4]
```

Add a new *instance* method `hello` to type `Foo`:

```
‣ class Foo
‣   def hello *args
‣      “Hello world: #{args.inspect}”
```

```
  end
end
Foo.new.hello :a1, :a2
  “Hello world: [:a1, :a2]”
```

Add a new *instance* method `good_bye` to an *instance* `foo1` of `Foo`:

```
foo1=Foo.new
def foo1.good_bye *args
  “Good bye: #{args.inspect}”
end
foo1.good_bye :b1, :b2
  “Good bye: [:b1, :b2]”
```

Add a new *instance* method `greetings` to an *instance* `foo1` of `Foo`, using the *singleton* class for `foo1`.

```
foo1=Foo.new
class << foo1
  def greetings *args
     “Greetings: #{args.inspect}”
  end
end
foo1.greetings :c1, :c2
  “Greetings: [:c1, :c2]”
```

Add a new *class* method `good_night` to type `Foo`:

```
def Foo.good_night *args
  “Good night: #{args.inspect}”
end
Foo.good_night :d1, :d2
  “Good night: [:d1, :d2]”
```

Method Wrapping

Alias an existing *instance* method `hello`, redefine the method, and delegate to the old method in type `Foo`:

```
class Foo
  alias_method :hello2, :hello
  def hello *args
     “{{{” + hello2(*args) + “}}}”
  end
end
Foo.new.hello :e1, :e2
  “{{{Hello world: [:e1, :e2]}}}”
```

Alias an existing *class* method `doit`, redefine the method, and delegate to the old method in type `Foo`:

```
Foo.class_eval do
  class << self
     alias_method :good_night2, :good_night
     def good_night *args
        “<<<” + good_night2(*args) + “>>>”
     end
  end
end
Foo.good_night :f1, :f2
  “<<<Good night: [:f1, :f2]>>>”
```

Ruby Metaprogamming Cheat Sheet (By Example)

Interpreting Messages to Objects

Using method_missing to Handle “Missing” Methods

Dynamically handle any unknown message sent to `Echo`; print the message name followed by the argument list.

```
class Echo
  def method_missing method_sym, *args
     p “#{method_sym}: #{args.inspect}”
  end
end
Echo.new.yell “Hello”, “world!”
Echo.new.say “Good”, “bye!”
```

```
“yell: [\”Hello\”, \”world!\”]”
“yell: [\”Good\”, \”Bye!\”]”
```

Evaluating Strings as Code

Add New Methods on Demand

Dynamically add a method for any unknown message sent to `Echo`.

```
class Echo
  def method_missing method_sym, *args
     p "defining method #{method_sym}"
     eval <<-EOF
        def #{method_sym.to_s} *args
           p "#{method_sym}: " +
              args.inspect
        end
     EOF
     send(method_sym, *args)
  end
end
Echo.new.yell "Hello", "world!"
Echo.new.yell "good", "bye"
```

```
“defining method yell”
“yell: [\”Hello\”, \”world!\”]”
“yell: [\”Good\”, \”bye!\”]”
```

Add an *instance* method for a single object of type `Echo` to the *singleton class* of the object. The name of the method is the value of `method_name`.

```
echo = Echo.new
method_name = “new_method”
sing = class << echo; self; end
sing.class_eval <<-EOF
  def #{method_name} *args
     p "#{method_name}: " +
        args.inspect
  end
EOF
```

```
nil
```

Add an *instance* method for a single object of type `Echo` to the *singleton class* of the object. The name of the method is the value of `method_name`. (Alternative approach)

```
▸ echo = Echo.new
▸ method_name = “new_method”
▸ echo.instance_eval <<-EOF
▸   def #{method_name} *args
▸      p "#{method_name}: " +
▸         args.inspect
▸   end
▸ EOF
   nil
```

Add an *instance* method to type `Echo` whose name is the value of `method_name`. It can be invoked by any instance of `Echo`.

```
▸ Echo.class_eval <<-EOF
▸   def #{method_name} *args
▸      p "#{method_name}: " +
▸         args.inspect
▸   end
▸ EOF
   nil
```

Assorted References

- *Ruby for Rails*, David Black, Manning.
- http://ola-bini.blogspot.com/2006/09/ruby-metaprogramming-techniques.html
- http://www.vanderburg.org/Speaking/Stuff/oscon05.pdf
- http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html
- http://poignantguide.net/dwemthy/
- http://weblog.jamisbuck.org/2006/4/20/writing-domain-specific-languages
- http://rubyquiz.com/metakoans.rb