Metaprogramming in Ruby

1.3.6 eval

The module **Kernel** has the **eval()** method and is used to execute code in a string. The **eval()** method can evaluate strings spanning many lines, making it possible to execute an entire program embedded in a string. **eval()** is slow - calling **eval()** effectively compiles the code in the string before executing it. But, even worse, **eval()** can be dangerous. If there's any chance that external data - stuff that comes from outside your application - can wind up inside the parameter to **eval()**, then you have a security hole, because that external data may end up containing arbitrary code that your application will blindly execute. ***eval()*** *is now considered a method of last resort*.

```
str = "Hello"
puts eval("str + ' Rubyist'") # => "Hello Rubyist"
```

1.3.7 instance_eval, module_eval, class_eval

instance_eval(), **module_eval()** and **class_eval()** are special types of **eval()**.

1.3.7.1 instance_eval

The class **Object** has an **instance_eval()** public method which can be called from a specific object. It provides access to the instance variables of that object. It can be called either with a block or with a string.

```
class Rubyist
  def initialize
    @geek = "Matz"
  end
end
obj = Rubyist.new
# instance_eval can access obj's private methods
# and instance variables
obj.instance_eval do
```

```
  puts self # => #<Rubyist:0x2ef83d0>
  puts @geek # => Matz
end
```

*The block that you pass to **instance_eval( )**helps you dip inside an object to do something in there. You can wreak havoc on encapsulation! No data is private data anymore.*

instance_eval can also be used to add *class methods* as shown below:

```
class Rubyist
end
Rubyist.instance_eval do
  def who
    "Geek"
  end
end
puts Rubyist.who # => Geek
```

Remember our example back on **1.1.5 Anonymous class #4**? We had used **instance_eval** there.

1.3.7.2 module_eval, class_eval

The **module_eval** and **class_eval** methods operate on modules and classes rather than on objects. The **class_eval** is defined as an alias of **module_eval**.

The **module_eval** and **class_eval** methods can be used to add and retrieve the values of class variables from ***outside*** *a class*.

```
class Rubyist
  @@geek = "Ruby's Matz"
end
puts Rubyist.class_eval("@@geek") # => Ruby's Matz
```

The **module_eval** and **class_eval** methods can also be used to add instance methods to a module and a class. In spite of their names, **module_eval** and **class_eval** are functionally identical and each may be used with ether a module or a class.

```
class Rubyist
end
Rubyist.class_eval do
  def who
    "Geek"
  end
end
obj = Rubyist.new
puts obj.who # => Geek
```

Note: **class_eval** defines instance methods, and **instance_eval** defines class methods.

1.3.8 class_variable_get, class_variable_set

To add or retrieve the values of class variables, the methods **class_variable_get** (this takes a symbol argument representing the variable name and it returns the variable’s value) and **class_variable_set** (this takes a symbol argument representing a variable name and a second argument which is the value to be assigned to the variable) can be used.

```
class Rubyist
  @@geek = "Ruby's Matz"
end
Rubyist.class_variable_set(:@@geek, 'Matz rocks!')
puts Rubyist.class_variable_get(:@@geek) # => Matz rocks!
```

1.3.9 class_variables

To obtain a list of class variable names as an array of strings, we can use the **class_variables** method.

```
class Rubyist
  @@geek = "Ruby's Matz"
  @@country = "USA"
end

class Child < Rubyist
  @@city = "Nashville"
end
print Rubyist.class_variables # => [:@@geek, :@@country]
puts
p Child.class_variables # => [:@@city]
```

You will observe from the program output that the method **Child.class_variables** gives us the class variables (**@@city**) defined in the class and not the inherited ones(**@@geek, @@country**).

1.3.10 instance_variable_get, instance_variable_set

One can retrieve the value of instance variables using the **instance_variable_get** method.

```
class Rubyist
  def initialize(p1, p2)
    @geek, @country = p1, p2
  end
end
obj = Rubyist.new('Matz', 'USA')
puts obj.instance_variable_get(:@geek) # => Matz
puts obj.instance_variable_get(:@country) # => USA
```

You can also add instance variables to classes and objects *after* they have been created using **instance_variable_set**.

```
class Rubyist
  def initialize(p1, p2)
    @geek, @country = p1, p2
  end
end
obj = Rubyist.new('Matz', 'USA')
puts obj.instance_variable_get(:@geek) # => Matz
puts obj.instance_variable_get(:@country) # => USA
obj.instance_variable_set(:@country, 'Japan')
puts obj.inspect # => #<Rubyist:0x2ef8038 @country="Japan", @g
```

1.3.11 const_get, const_set

One can similarly get and set constants using **const_get** and **const_set**.

const_get returns the value of the named constant, as shown below:

```
puts Float.const_get(:MIN) # => 2.2250738585072e-308
```

const_set sets the named constant to the given object, returning that object. It creates a new constant if no constant with the given name previously existed, as shown below:

```
class Rubyist
end
puts Rubyist.const_set("PI", 22.0/7.0) # => 3.14285714285714
```

As **const_get** returns the value of a constant, you could use this method to get the value of a class name and then append the new method to create a new object from that class. This could even give you a way of creating objects at runtime by prompting the user to enter class names and method names. One can create a completely new class at runtime by using **const_set**.

```
# Let us call our new class 'Rubyist'
# (we could have prompted the user for a class name)
class_name = "rubyist".capitalize
Object.const_set(class_name, Class.new)
# Let us create a method 'who'
# (we could have prompted the user for a method name)
class_name = Object.const_get(class_name)
puts class_name # => Rubyist
class_name.class_eval do
  define_method :who do |my_arg|
    my_arg
  end
end
obj = class_name.new
puts obj.who('Matz') # => Matz
```

1.4 Bindings

Entities like local variables, instance variables, self. . . are basically *names bound to objects*. We call them **bindings**.

1.5 Ruby Blocks and Bindings

A Ruby block contains both the code and a set of bindings. When you define a block, it simply grabs the bindings that are there at that moment, then it carries those bindings along when you pass the block into a method.

```
def who
  person = "Matz"
  yield("rocks")
end
person = "Matsumoto"
who do |y|
  puts("#{person}, #{y} the world") # => Matsumoto, rocks the
  city = "Tokyo"
end
# puts city # => undefined local variable or method 'city' for
```

Observe that the code in the block sees the **person** variable that was around when the block was defined, *not the* method's **person** variable. Hence a block captures the local bindings and carries them along with it. You can also define additional bindings inside the block, but they disappear after the block ends.

<Home | Prev | Next>

Note: The material in these study notes is drawn primarily from the references mentioned on the last page. Our acknowledgment and thanks to all of them.
This page was last updated on 1st Sept. 2011.