Metaprogramming in Ruby

1.6 Solved Problems

1.6.1 Problem 1

This example has been adapted from Dave Thomas' screencast "Episode 5: Nine Examples of Metaprogramming".

We all know that the Core Ruby course at RubyLearning.org runs for 8 weeks. Every week there is a quiz and marks are allocated out of 10. At the end of 8 weeks the student can find out his percentage score. For example, if a students scores 5,10,10,10,10,10,10,10 marks in 8 weeks i.e. his percentage score is 93.75%

Problem Statement: Every Core Ruby batch has hundreds of students. Let us assume that we have a Ruby method that does this percentage calculation and returns the same value given the same set of arguments. We don't need to go on calculating the value each time. We only need to calculate the value the first time and then somehow associate that value with that set of arguments. Then the next time it gets called, if we have the same arguments, we can use the previously stored value as the return value of this method thus bypassing the need to do the calculations again. *We need to develop a solution to address this problem using Metaprogramming techniques*.

1.6.1.1 Existing class and method

To start with, let us look at the existing class and method and then keep modifying it to achieve the above result.

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
```

```
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

r = Result.new
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

In the code above, we have a **Result** class and a **total** method that takes a list of **scores** per student. The **scores** represent the marks obtained by a student in each of the 8 quizzes in the course. The private method **percentage_calculation** does the actual percentage calculation. To test this, we call the **total** method 4 times. The first two and the last two have the same set of scores. When we run our code we get the following output:

```
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
```

Looking at the above output, we realize that we have called the **total** method 4 times and that in turn also called the **percentage_calculation** method 4 times. We are now going to try and reduce the number of calls to the **percentage_calculation** method.

1.6.1.2 Normal Solution

One way to reduce the number of calls to the **percentage_calculation** method is to somehow store the previous results in memory. For this, we shall define a subclass named **MemoResult** that has a hash named **@mem** and then use the **@mem** hash in the **total** method, as shown in the modified code below:

```
class Result
```

```
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

class MemoResult < Result
  def initialize
    @mem = {}
  end
  def total(*scores)
    if @mem.has_key?(scores)
      @mem[scores]
    else
      @mem[scores] = super
    end
  end
end

r = MemoResult.new
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

The **Hash** class has a **has_key?** method that returns **true** if the given key is present in **@mem**. In the above program, if **has_key?** is **true** then we return the value available in **@mem** for that key otherwise we do the calculation by calling **percentage_calculation(*scores)** and storing the value in **@mem**. Let us see the output:

```
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
93.75
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
100.0
```

Observe that we have saved calling the **percentage_calculation(*scores)** method for the second and fourth call to **r.total**.

1.6.1.3 Solution using Class.new and define_method

The **MemoResult** class above, is intimately tied to its parent **Result** class. To avoid that, let us generate this subclass dynamically using whetever we have learnt so far in **Ruby Metaprogramming**.

To do that, let us write a method called **mem_result** that takes two parameters: the name of the parent class and the name of a method (the method will return the name of the class). Here's the code:

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

def mem_result(klass, method)
  mem = {}
  Class.new(klass) do
    define_method(method) do |*args|
      if mem.has_key?(args)
        mem[args]
      else
        mem[args] = super
      end
    end
  end
end

r = mem_result(Result, :total).new
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

The output is:

```
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
93.75
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
100.0
```

The code **Class.new(klass)** creates a new anonymous class with the given superclass **klass**. The block is used as the body of the class and contains the methods in that class. The **define_method** defines the method **method** (which is the second argument to **mem_result**). This takes the **method** arguments in **args**.

Note: We have done away with the **initialize** method and the instance variable **@mem**. Instead we use a local variable **mem** since the block is a closure and this local variable **mem** is available inside the block.

1.6.1.4 Solution using anonymous class

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

r = Result.new

# Anonymous class on object
def r.total(*scores)
  @mem ||= {}
  if @mem.has_key?(scores)
    @mem[scores]
  else
    @mem[scores] = super
  end
end

puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

1.6.1.5 Solution using an anonymous class created on the fly

```
class Result
  def total(*scores)
```

```
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

def mem_result(obj, method)
  obj.class.class_eval do
    mem ||= {}
    define_method(method) do |*args|
      if mem.has_key?(args)
        mem[args]
      else
        mem[args] = super
      end
    end
  end
end

r = Result.new
mem_result(r, :total)

puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

In the above code, we have written a new **mem_result** method that takes as an argument an object (**obj**) for which an anonymous class needs to be generated and the second argument being the method (**method**) to be created within this anonymous class.

We had previously used **define_method** to create a method on the fly. The problem is that **define_method** is only defined on classes and modules and what we have here is an object. Hence we get the class of the object by **obj.class** and then use the **class_eval** and **define_method** methods on that class to add an instance method (**method**) to the class. Let us run the code and check the output.

```
result.rb:21:in `total': super: no superclass method `total' (
  from result.rb:30
```

The code does not run.

The line **mem[args] = super** is trying to call the **total** method of the class **Result**, from the anonymous class. The problem is that we have defined our **total** method directly in class **Result**. We have said **obj.class** which is class **Result** and that's not going to work. What we need to do is create an anonymous class and put our **total** method in this anonymous class. Also, our anonymous class needs to be a subclass of our class **Result**.

Let us create our anonymous class as follows:

```
  anon = class << obj
    self
  end
```

self above gives us our anonymous class object. This object is then being referenced by our variable **anon**. Most rubyists would put this code in one line to indicate that they are extracting the ghost class, as follows:

```
anon = class << obj; self; end
```

Having got our anonymous class object, we shall use it in our **class_eval** method, as shown in the code below:

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

def mem_result(obj, method)
  anon = class << obj; self; end
  anon.class_eval do
    mem ||= {}
    define_method(method) do |*args|
      if mem.has_key?(args)
        mem[args]
      else
```

```
        mem[args] = super
      end
    end
  end
end

r = Result.new
mem_result(r, :total)

puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

Our code runs successfully, giving us the desired result.

<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 16th Dec. 2009.