(w) http://www.ruby-software.com/
(e) sales@ruby-software.com
(p) +1 770-325-4352

Development and Consulting

canvas

Canvas LMS

pentaho™
open source business intelligence™

pentaho business intelligence

Open Source Experts

Usage of "send" method

Last but not the least is the "send" (instance method) method, which enables Ruby objects to invoke methods belonging to an object providing the necessary arguments. This is especially useful when you would want to invoke methods dynamically based on an input during runtime. See Listing 7.

```
Chef.presentation("lambchops") {puts "Lambchops decorated"}
chef = Chef.new
chef.lambchops
```

The above mentioned is an implementation utilizing both the `define_method` and send method in Ruby to bring another level of dynamism where you can even provide the block depending on what you would like the newly created method to do from outside the class.

Some practical uses of Metaprogramming

Apart from uses mentioned earlier (Artificial Intelligence and Reduced lines of code), Metaprogramming has found immense use in Testing Frameworks. When certain behaviors need to be tested, we would have to dynamically modify certain methods or add methods in-order to set expectations on them. This is only possible through Metaprogramming, where we will be able to inject the mock object or class with the necessary methods and get the expected results.

See Listing 8 and Listing 9.

The code above has methods `christmas_theme`, `thanksgiving_theme` and `newyear_theme` which have similar names. Using `define_method` makes the code DRY and reduces the number of lines of code considerably.

Conclusion

In conclusion, Metaprogramming may not be the ultimate solution to all your programming needs, I hope, however, thatthis article left you with a solid understanding of the basic concept so that you will be able to identify situations or scenarios when this will be the best option, thereby increasing the performance of your application and reducing overhead costs!

AUTHOR SYNOPSIS

Anjana Nair works for Ruby Software – a Ruby on Rails development and consulting firm. She is a mentor and lead for Ruby On Rails development at Ruby Software. JQuery and ROR are some of her favorite programming languages! When she is not working, she likes to spend time with her daughter Gowri who is 9 but is looking forward to being a die-hard Ruby On Rails developer one day! She can be reached at Anjana@ruby-software.com

Listing 9. *Repetitive code (methods with similar names doing similar things) re-factored to become DRY using Metaprogramming*

```
class Chef
   def cook
      puts "cook delicious food"
   end
   def christmas_theme(menuitem)
      puts "#{menuitem} decorated"
   end
   def newyear_theme(menuitem)
      puts "#{menuitem} decorated"
   end
   def thanksgiving_theme(menuitem)
      puts "#{menuitem} decorated"
   end
###### The above code re-factored using
                  Metaprogramming to make it DRY
                  #####
=begin
   class << self
      def create_presentation_theme(occasion)
         define_method("#{occasion}_theme") do |menu_
                                                item|
            puts "#{menu_item} decorated"
         end
      end
   end
=end
end
Chef.create_presentation_theme("christmas")
   chef = Chef.new
   chef.christmas_theme("Cookies") # => Cookies
                  decorated chef.christmas_
                  theme("Cake")  # => Cake decorated
Chef.create_presentation_theme("thanksgiving")
   chef = Chef.new
   chef.thanksgiving _theme("Turkey") # => Turkey
                  decorated
         chef.thanksgiving _theme("Mashed
                  potatoes")  # => Mashed potatoes
                  decorated
```

conceptually and for understanding purposes we can say that meta-classes are associated with the Class itself while singletons get associated with the objects of the class). Take a look at this: Listing 4. Here, we have actually added the method `create_presentation_theme` to the meta-class of Chef. And we have programmed the meta-class to set the Christmas theme for menu items: "Cookies" and "Cake".

Note: The advantage of using define-method is it reduces the number of lines of code. The picture below illustrates the meta-class and methods it has in it (Figure 1).

Using "instance_eval" to Metaprogram

Ruby also provides us with the `instance_eval` method which is another way of achieving the functionality above. Let's take a look: Listing 5.

Metaprogramming using "class_eval"

Now let us explore `class_eval` where we can add new methods to the current object and other instances dynamically. Let's take a look at the code that explains this: Listing 6.

Listing 6. *Example of adding method using "class_eval"*

```
class Chef
   def cook
      puts "cook delicious food"
   end
end
   chef = Chef.new
   Chef.class_eval {
      def presentation(menuitem)
         puts "#{menuitem} decorated"
      end
   }

   chef.presentation("Lambchops") # => "Lambchops
                    decorated"
   chef.presentation("Turkey") # => " Turkey
                    decorated"
```

Listing 7. *Example of method invocation using "send"*

```
 class Chef
   def cook
      puts"cook delicious food"
   end

   class << self
      def presentation(menuitem, &block)
         self.send(:define_method, menuitem, &block)

      end
   end
end
```

Listing 8. *Example of method stubbing using Metaprogramming*

```
 class Chef
  attr_accessor :certified
   def cook
    if certified
    "Cooks delicious food"
    else
     "Is not qualified to cook"
    end
  end
end
# Testing the above code using RSpec
describe "Chef" do
  before :each do
    @chef = Chef.new
  end
  it "should be certified and can cook delicious food"
                    do
    Chef.class_eval{
      def certified
        true
      end
    }
    @chef.cook.should == "Cooks delicious food"
  end

    it "is not certified and cannot cook well" do
    Chef.class_eval{
      def certified
        false
      end
    }
    @chef.cook.should match(/Is not qualified/)
  end
end
```

guin. This functionality won't be available to any other instance of Penguin.

Now that we've understood what a singleton class is, let us try to continue our research and dig further into Metaprogramming by understanding a meta-class. At this point let us switch back to our earlier example of the restaurant chef and continue with the rest of the illustration.

In this scenario, the chef was "programmed" to cook but management added a touch of dynamism to the current "function" by equipping him with the ability to present it well too, hereby opening the doors to making higher revenue for the company.

Metaprogramming using "define_method"

Let us define a class "Chef" (see below). The chef does his job of cooking (defined by the method "cook").

```
class Chef
   def cook
      puts "Cook yummy food!"
   end
end
```

Now let us bring in the functionality to "present" it well too (with the method called "presentation"). See Listing 3.

All Ruby classes are provided with a private method called `define_method` which enables a class to expand its methods dynamically.

In the example code sample above "presentation" is the newly added instance method of the "Chef" class.

The dynamically created "presentation" method is available to all objects of this class and we can set the presentation for any kind of menu item as the method "presentation" takes in an argument => menu_item so that it knows what kind of decoration is required for that menu item.

Meta-class

Now, let 's see if we can set a presentation theme based on an occasion like "Christmas", "New Year", "Thanksgiving" etc for different menu items. Here we'll be bringing the code to another level of abstraction where we'll introduce meta-classes and be able to control the functionality from outside the class. (Meta-classes and singleton-classes are used interchangeably in Ruby. There's no hard distinction between the both, however

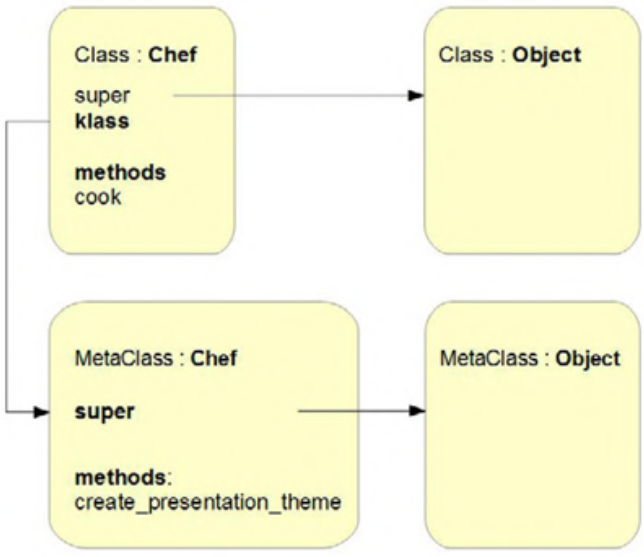

Figure 1. *Overview of Meta-class methods and its relationship with its class*

Listing 5. *Example of adding method using "instance_eval"*

```
class Chef
   def cook
      puts"cook delicious food"
   end
end

Chef.instance_eval {
   def create_presentation_theme(occasion)
      define_method("#{occasion}_theme") do |menuitem|
         puts "#{menuitem} decorated for #{occasion}"
      end
   end
}

Chef.create_presentation_theme("christmas")

   chef = Chef.new
   chef.christmas_theme("Cookies") # => Cookies
                  decorated for christmas
   chef.christmas_theme("Cake") # => Cake decorated
                  for christmas

Chef.create_presentation_theme("thanksgiving")

   chef = Chef.new
   chef.thanksgiving_theme("Turkey") # => Turkey
                  decorated for thanksgiving
   chef.thanksgiving_theme("Salad") # => Salad
                  decorated for thanksgiving

In the code above, just like in the previous example
                  we have dynamically added the
                  "create_presentation_theme" class
                  method to the meta-class "Chef".
```

Explaining it a little further: It's the process of adding, updating or removing code or any number of methods from your program dynamically. This level of programming helps to isolate dependencies and avoid inefficient code that can arise in the form of repetitive code thereby keeping code clean and DRY.

Now, how is this possible?

Before you delve further into the Metaprogramming concept and understand how we can add methods in Ruby dynamically, you should understand the concept of self and meta-classes.

This is important because Metaprogramming in short is programming the "self", the "meta-class" or the "singleton-class" dynamically.

Ruby's method look-up

Let's start with trying to figure out what happens when you call a method in Ruby so that you can then get in there and do your magic! See the following example cod.

Bird is a parent class of Duck and Penguin. Penguins cannot fly and ducks "quack". These are the natural mannerisms of the classes Penguin and Duck. Let us see how we can call methods on these classes (Listing 1).

In the code sample above we're first making an attempt to invoke the `make_sound()` method. At this point the code first checks to see if the object "duck" knows about this method.

The duck object's singleton class remains invisible until you start adding methods to it. At present we haven't added any methods to it and it stays hidden. We'll elaborate on this in the next example.

When it doesn't find a method in its singleton class, it looks for it in its parent class which is "Duck" and there it finds a match!

Now let's take the second method call into consideration – `duck.move`. Here Ruby doesn't find a match within the current object "duck", neither does it find a match in its parent class "Duck", but will find a match in the super class "Bird". As we increase the chain of successors for the "Bird" class we are inadvertently adding an additional "method look-up" action to find a match for a method call. As it traverses through the singleton class first then the parent class and then the parent's parent class it could also hit a point where it cannot find a method when it throws a `method_missing` error. This method_missing method can be also be programmed to handle errors gracefully (by customizing it to behave differently).

Now let us see how we can add functionality dynamically to certain objects or instances only, using singleton class programming which is another aspect of Metaprogramming.

Metaprogramming Singleton-classes

In the example (Listing 2) we've added a method to the object "penguin". This method (`lives_in`) belongs to the "singleton class" of the object "penguin". Now when you invoke the method `lives_in`, it looks for it in singleton class first and will find it there and thus will not need to dig further in the inheritance chain to find the method. Here we've actually programmed an instance of Pen-

Listing 3. *Example of adding method using "define_method"*

```
class Chef
  def cook
     puts "Cook delicious food"
  end

  define_method("presentation") do |menu_item|
     puts "#{menu_item} decorated"
  end
end
  chef = Chef.new
  chef.presentation("Lambchops") # => "Lambchops
                    decorated"
  chef.presentation("Turkey") # => " Turkey
                    decorated"
We have just added a new method called "presentation"
                    to "Chef".
```

Listing 4. *Example of adding method to the Meta-class*

```
class Chef
  def cook
     puts "cook delicious food"
  end
  class << self
     def create_presentation_theme(occasion)
        define_method("#{occasion}_theme") do |menu_
                    item|
           puts "#{menu_item} decorated"
        end
     end
  end
end
Chef.create_presentation_theme("christmas")
  chef = Chef.new
  chef.christmas_theme("Cookies") # => Cookies
                    decorated chef.christmas_
                    theme("Cake")  # => Cake decorated
```

Metaprogramming in Ruby

Metaprogramming has been around long before Ruby, but for some programmers it still remains an unsolved mystery. You will see in this article that it is nothing but programming programs to do wonderful things!

Ruby stands out from other languages like Java and C due to the flexibility and robustness it offers in Metaprogramming. The importance of Metaprogramming can be gauged from its large number of applications. These include the areas of Artificial Intelligence and Reflection, since it allows us to manipulate programs dynamically. This article should help you to have a solid understanding of what Metaprogramming is, realize its simplicity and the immense possibilities it can bring to you.

Let's explain the usage scenario with a real-life example

A chef working in a restaurant is expected to cook delicious food and deliver it with a lovely presentation.

Now let's assume that the chef only knows how to cook with the right recipe and doesn't have the ability or skills to be able to decorate the platter. The management will definitely have to improvise on his skill-set and enrich him with "food presentation" skills. So the chef gets subsequently trained in "presentation" and is now decked with 2 qualities: "cook well" and "present it well" while on the job!

Similarly, if a need arises to add more capabilities to the subject in question, that is "The Chef", it's just a question of enriching the "self" ("Chef") further by adding additional functionalities while on the job.

This is similar to the concept of Metaprogramming but explained in our day to day life – enriching the chef's "self".

Listing 1. *Method look-up through Inheritance*

```
class Bird
   def make_sound
      puts "Tweet"
   end
   def move
      puts "Fly"
   end
end

class Duck < Bird
   def make_sound
      puts "Quack"
   end
end

class Penguin < Bird
   def move
      puts "Waddle"
   end
end

duck = Duck.new
duck.make_sound # => Quack
duck.move # =>Fly

penguin = Penguin.new
penguin.move # =>Waddle
```

Listing 2. *Example of adding method to singleton class*

```
penguin = Penguin.new
def penguin.lives_in
   puts "I live in Antarctica"
end
penguin.lives_in #=>  I live in Antarctica
```