CSE M226: Programming Languages

Lecture 8:Metaprogramming

Rokan Uddin Faruqui

Dept. of Computer Science and Engineering
University of Chittagong, Bangladesh
Email: rufaruqui@cu.ac.bd

Outline

- What is Metaprogramming?
- Objects and Classes
- Singletons
- Inheritance and Visibility
- Modules and Mixins
- Metaprogramming Class-Level Macros
- Different forms of Class Definition
- Hook Methods

Metaprogramming

Metaprogramming– writing code that writes code.

- Programming is all about building layers of abstractions
- Some programming languages – such as *C* – are close to the machine.
- The distance from *C* code to the application domain can be large.
- Other languages – *Ruby*, perhaps – provide higher-level abstractions and hence let you start coding closer to the target domain.
- For this reason, most people consider a higher-level language to be a better starting place for application development

Class-Level Macros

```
1
      class Song
3       attr_accessor :duration
      end
5
      class Album < ActiveRecord::Base
7       has_many :tracks
      end
9
```

Class-Level Macros

```
class Logger
    def self.add_logging
      def log(msg)
        STDERR.puts Time.now.strftime("%H:%M:%S: ")
    + "#{self} (#{msg})"
      end
    end
 end
class Example < Logger
      add_logging
end
ex = Example.new
ex.log("hello")

produces:
12:31:38: #<Example:0x007fcc5c0473d0> (hello)
```

Class-Level Macros

```
class Song < Logger
  add_logging "Song"
end

class Album < Logger
  add_logging "CD"
end
```

Class-Level Macros

```
class Logger
        def self.add_logging(id_string)
            define_method(:log) do |msg|
              now = Time.now.strftime("%H:%M:%S")
              STDERR.puts "#{now}-#{id_string}: #{
    self} (#{msg})"
            end
        end
end

class Song < Logger
    add_logging "Tune"
end

class Album < Logger
    add_logging "CD"
```

Class-Level Macros and Modules

```
module AttrLogger
  def attr_logger(name)
    attr_reader name
    define_method("#{name}=") do |val|
      puts "Assigning #{val.inspect} to #{name}"
      instance_variable_set("@#{name}", val)
    end
  end
end

class Example
  extend AttrLogger
  attr_logger :value
end
```

Struct.new

```
Person = Struct.new(:name, :address, :likes)
 class Person
  def to_s
    "#{self.name} lives in #{self.address} and
    likes #{self.likes}"
  end
 end

```

Struct.new

```
class Person < Struct.new(:name, :address, :likes)
  def to_s
    "#{self.name} lives in #{self.address} and likes #{self.likes}"
  end
end

```

Hook Methods

- **Hook method** – is a method that you write but that Ruby calls from within the interpreter when some particular event occurs.
- The interpreter looks for these methods by name – if you define a method in the right context with an appropriate name – Ruby will call it when the corresponding event happens.
- **included** is an example of a hook method (sometimes called a **callback**)

The inherited Hook

If a class defines a class method called *inherited*, Ruby will call it whenever that class is subclassed. This hook is often used in situations where a base class needs to keep track of its children. For Example,

- an online store might offer a variety of shipping options.
- each might be represented

The inherited Hook

If a class defines a class method called *inherited*, Ruby will call it whenever that class is subclassed. This hook is often used in situations where a base class needs to keep track of its children. For Example,

- an online store might offer a variety of shipping options.
- each might be represented by a separate clas – a subclass of a single Shipping class.

The inherited Hook

If a class defines a class method called *inherited*, Ruby will call it whenever that class is subclassed. This hook is often used in situations where a base class needs to keep track of its children. For Example,

- an online store might offer a variety of shipping options.
- each might be represented by a separate clas – a subclass of a single Shipping class.
- this parent class could keep track of all the various shipping options by recording every class that subclasses it.

The inherited Hook

If a class defines a class method called *inherited*, Ruby will call it whenever that class is subclassed. This hook is often used in situations where a base class needs to keep track of its children. For Example,

- an online store might offer a variety of shipping options.
- each might be represented by a separate clas – a subclass of a single Shipping class.
- this parent class could keep track of all the various shipping options by recording every class that subclasses it.
- when it comes time to display the shipping options to the user, the application could call the base class, asking it for a list of its children

The inherited Hook

If a class defines a class method called *inherited*, Ruby will call it whenever that class is subclassed. This hook is often used in situations where a base class needs to keep track of its children. For Example,

- an online store might offer a variety of shipping options.
- each might be represented by a separate clas – a subclass of a single Shipping class.
- this parent class could keep track of all the various shipping options by recording every class that subclasses it.
- when it comes time to display the shipping options to the user, the application could call the base class, asking it for a list of its children

The inherited Hook

```
class Shipping # Base class
 @children = []
 def self.inherited(child)
  @children << child
 end
 def self.shipping_options(weight, international)
   @children.select {|child| child.can_ship(weight,
     international)}
 end
end

class MediaMail < Shipping
 def self.can_ship(weight, international)
   !international
  end
end
```

The inherited Hook

```
1  class FlatRatePriorityEnvelope < Shipping
     def self.can_ship(weight, international)
3      weight < 64 && !international
     end
5  end

7  class InternationalFlatRateBox < Shipping
     def self.can_ship(weight, international)
9      weight < 9*16 && international
     end
11 end
```

The inherited Hook

```
puts "Shipping 16oz domestic"
puts Shipping.shipping_options(16, false)
puts "\nShipping 90oz domestic"
puts Shipping.shipping_options(90, false)
puts "\nShipping 16oz international"
puts Shipping.shipping_options(16, true)

```

The inherited Hook

produce

```
    Shipping 16oz domestic
    MediaMail
    FlatRatePriorityEnvelope
    Shipping 90oz domestic
    MediaMail
    Shipping 16oz international
    InternationalFlatRateBox

```

The method_missing Hook

- Ruby executes a method call by looking for the method, first in the object?s class, then in its superclass, then in that class's superclass, and so on.
- If the method is not found by the time we run out of **superclasses** then Ruby tries to invoke the hook method *method_missing* on the original object.
- The key here is that *method_missing* is simply a Ruby method. We can override it in our own classes to handle calls to otherwise undefined methods in an application-specific way.

```
def method_missing(name, *args, &block)
```

method_missing as a filter

- dynamic finder facility in the Ruby on Rails ActiveRecord module

```
1  pickaxe = Book.find_by_title("Programming Ruby")
   daves_books = Book.find_all_by_author("Dave
     Thomas")
3
```

method_missing as a filter

- Active Record does not predefine all these potential finder methods. Instead, it uses our old friend method_missing.
- Inside that method, it looks for calls to undefined methods that match the pattern **/^find_(all_)?by_(.*)/**
- If the method being invoked does not match this pattern or if the fields in the method name don?t correspond to columns in the database table,
- Active Record calls super so that a genuine method_missing report will be generated.