Extracted from:

Metaprogramming Ruby

The Pragmatic Programmers

Metaprogramming Ruby

Paolo Perrotta

Edited by Jill Steinberg

The Facets of Ruby Series

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-47-6

ISBN-13: 978-1-934356-47-0

Printed on acid-free paper.

P1.0 printing, January 2010

Version: 2010-1-29

Whenever someone says they have "a cool trick," take them outside and slap them up.

► Jim Weirich

Appendix C

Spell Book

This appendix is a "spell book"—a quick reference to all the "spells" in the book, in alphabetical order. Most of these spells are metaprogramming related (but the ones from Appendix A, on page 242, are arguably not that "meta").

Each spell comes with a short example and a reference to the page where it's introduced. Go to the associated pages for extended examples and the reasoning behind each spell.

C.1 The Spells

Argument Array

Collapse a list of arguments into an array.

```
def my_method(*args)
  args.map {|arg| arg.reverse }
end

my_method('abc', 'xyz', '123') # => ["cba", "zyx", "321"]
```

For more information, see page 248.

Around Alias

Call the previous, aliased version of a method from a redefined method.

```
class String
  alias :old_reverse :reverse

  def reverse
    "x#{old_reverse}x"
  end
end
```

```
"abc".reverse # => "xcbax"
```

For more information, see page 157.

Blank Slate

Remove methods from an object to turn them into *Ghost Methods (75)*.

```
class C
  def method_missing(name, *args)
    "a Ghost Method"
  end
end

obj = C.new
obj.to_s # => "#<C:0x357258>"

class C
  instance_methods.each do |m|
    undef_method m unless m.to_s =~ /method_missing|respond_to?|^__/
  end
end

obj.to_s # => "a Ghost Method"
```

For more information, see page 86.

Class Extension

Define class methods by mixing a module into a class's eigenclass (a special case of *Object Extension (153)*).

```
class C; end

module M
  def my_method
    'a class method'
  end
end

class << C
  include M
end

C.my_method # => "a class method"
```

For more information, see page 153.

Class Extension Mixin

Enable a module to extend its includer through a *Hook Method (183)*.

```
module M
  def self.included(base)
    base.extend(ClassMethods)
  end

  module ClassMethods
    def my_method
      'a class method'
    end
  end
end

class C
  include M
end

C.my_method # => "a class method"
```

For more information, see page 187.

Class Instance Variable

Store class-level state in an instance variable of the Class object.

```
class C
  @my_class_instance_variable = "some value"

  def self.class_attribute
    @my_class_instance_variable
  end
end

C.class_attribute # => "some value"
```

For more information, see page 129.

Class Macro

Use a class method in a class definition.

```
class C; end

class << C
  def my_macro(arg)
    "my_macro(#{arg}) called"
  end
end

class C
  my_macro :x # => "my_macro(x) called"
end
```

For more information, see page 138.

Clean Room

Use an object as an environment in which to evaluate a block.

```
class CleanRoom
  def a_useful_method(x); x * 2; end
end

CleanRoom.new.instance_eval { a_useful_method(3) }      # => 6
```

For more information, see page 109.

Code Processor

Process *Strings of Code (165)* from an external source.

```
File.readlines("a_file_containing_lines_of_ruby.txt").each do |line|
  puts "#{line.chomp} ==> #{eval(line)}"
end

# >> 1 + 1 ==> 2
# >> 3 * 2 ==> 6
# >> Math.log10(100) ==> 2.0
```

For more information, see page 166.

Context Probe

Execute a block to access information in an object's context.

```
class C
  def initialize
    @x = "a private instance variable"
  end
end

obj = C.new
obj.instance_eval { @x } # => "a private instance variable"
```

For more information, see page 107.

Deferred Evaluation

Store a piece of code and its context in a proc or lambda for evaluation later.

```
class C
  def store(&block)
    @my_code_capsule = block
  end

  def execute
    @my_code_capsule.call
  end
end
```

```
obj = C.new
obj.store { $X = 1 }
$X = 0

obj.execute
$X # => 1
```

For more information, see page 110.

Dynamic Dispatch

Decide which method to call at runtime.

```
method_to_call = :reverse
obj = "abc"

obj.send(method_to_call) # => "cba"
```

For more information, see page 66.

Dynamic Method

Decide how to define a method at runtime.

```
class C
end

C.class_eval do
  define_method :my_method do
    "a dynamic method"
  end
end

obj = C.new
obj.my_method # => "a dynamic method"
```

For more information, see page 70.

Dynamic Proxy

Forward to another object any messages that don't match a method.

```
class MyDynamicProxy
  def initialize(target)
    @target = target
  end

  def method_missing(name, *args, &block)
    "result: #{@target.send(name, *args, &block)}"
  end
end

obj = MyDynamicProxy.new("a string")
obj.reverse # => "result: gnirts a"
```

For more information, see page 80.

Flat Scope

Use a closure to share variables between two scopes.

```
class C
  def an_attribute
    @attr
  end
end

obj = C.new
a_variable = 100

# flat scope:
obj.instance_eval do
  @attr = a_variable
end

obj.an_attribute # => 100
```

For more information, see page 105.

Ghost Method

Respond to a message that doesn't have an associated method.

```
class C
  def method_missing(name, *args)
    name.to_s.reverse
  end
end

obj = C.new
obj.my_ghost_method # => "dohtem_tsohg_ym"
```

For more information, see page 75.

Hook Method

Override a method to intercept object model events.

```
$INHERITORS = []

class C
  def self.inherited(subclass)
    $INHERITORS << subclass
  end
end

class D < C
end
```

```
class E < C
end

class F < E
end

$INHERITORS # => [D, E, F]
```

For more information, see page 183.

Kernel Method

Define a method in module Kernel to make the method available to all objects.

```
module Kernel
  def a_method
    "a kernel method"
  end
end

a_method # => "a kernel method"
```

For more information, see page 53.

Lazy Instance Variable

Wait until the first access to initialize an instance variable.

```
class C
  def attribute
    @attribute = @attribute || "some value"
  end
end

obj = C.new
obj.attribute # => "some value"
```

For more information, see page 246.

Mimic Method

Disguise a method as another language construct.

```
def BaseClass(name)
  name == "string" ? String : Object
end

class C < BaseClass "string" # a method that looks like a class
  attr_accessor :an_attribute # a method that looks like a keyword
end

obj = C.new
obj.an_attribute = 1  # a method that looks like an attribute
```

For more information, see page 243.

Monkeypatch

Change the features of an existing class.

```
"abc".reverse # => "cba"

class String
  def reverse
    "override"
  end
end

"abc".reverse # => "override"
```

For more information, see page 35.

Named Arguments

Collect method arguments into a hash to identify them by name.

```
def my_method(args)
  args[:arg2]
end

my_method(:arg1 => "A", :arg2 => "B", :arg3 => "C") # => "B"
```

For more information, see page 247.

Namespace

Define constants within a module to avoid name clashes.

```
module MyNamespace
  class Array
    def to_s
      "my class"
    end
  end
end

Array.new # => []
MyNamespace::Array.new # => my class
```

For more information, see page 43.

Nil Guard

Override a reference to nil with an "or."

```
x = nil
y = x || "a value" # => "a value"
```

For more information, see page 246.

Object Extension

Define Singleton Methods by mixing a module into an object's eigenclass.

```
obj = Object.new

module M
  def my_method
    'a singleton method'
  end
end

class << obj
  include M
end

obj.my_method # => "a singleton method"
```

For more information, see page 153.

Open Class

Modify an existing class.

```
class String
  def my_string_method
    "my method"
  end
end

"abc".my_string_method # => "my method"
```

For more information, see page 33.

Pattern Dispatch

Select which methods to call based on their names.

```
$x = 0

class C
  def my_first_method
    $x += 1
  end

  def my_second_method
    $x += 2
  end
end

obj = C.new
obj.methods.each do |m|
  obj.send(m) if m.to_s =~ /^my_/
end
```

```
$x # => 3
```

For more information, see page 69.

Sandbox

Execute untrusted code in a safe environment.

```
def sandbox(&code)
  proc {
    $SAFE = 2
    yield
  }.call
end

begin
  sandbox { File.delete 'a_file' }
rescue Exception => ex
  ex   # => #<SecurityError: Insecure operation `delete' at level 2>
end
```

For more information, see page 174.

Scope Gate

Isolate a scope with the **class**, **module**, or **def** keyword.

```
a = 1
defined? a # => "local-variable"

module MyModule
  b = 1
  defined? a # => nil
  defined? b # => "local-variable"
end

defined? a  # => "local-variable"
defined? b  # => nil
```

For more information, see page 102.

Self Yield

Pass **self** to the current block.

```
class Person
  attr_accessor :name, :surname

  def initialize
    yield self
  end
end
```

```
joe = Person.new do |p|
  p.name = 'Joe'
  p.surname = 'Smith'
end
```

For more information, see page 250.

Shared Scope

Share variables among multiple contexts in the same *Flat Scope (105)*.

```
lambda {
  shared = 10

  self.class.class_eval do
    define_method :counter do
      shared
    end

    define_method :down do
      shared -= 1
    end
  end
}.call

counter           # => 10
3.times { down }
counter           # => 7
```

For more information, see page 106.

Singleton Method

Define a method on a single object.

```
obj = "abc"

class << obj
  def my_singleton_method
    "x"
  end
end

obj.my_singleton_method # => "x"
```

For more information, see page 135.

String of Code

Evaluate a string of Ruby code.

```
my_string_of_code = "1 + 1"
eval(my_string_of_code) # => 2
```

For more information, see page 165.

Symbol To Proc

Convert a symbol to a block that calls a single method.

```
[1, 2, 3, 4].map(&:even?)   # => [false, true, false, true]
```

For more information, see page 253.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Metaprogramming Ruby's Home Page
http://pragprog.com/titles/ppmetr
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/ppmetr.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)