Extracted from:

Metaprogramming Ruby 2

Program Like the Ruby Pros

The Pragmatic Bookshelf

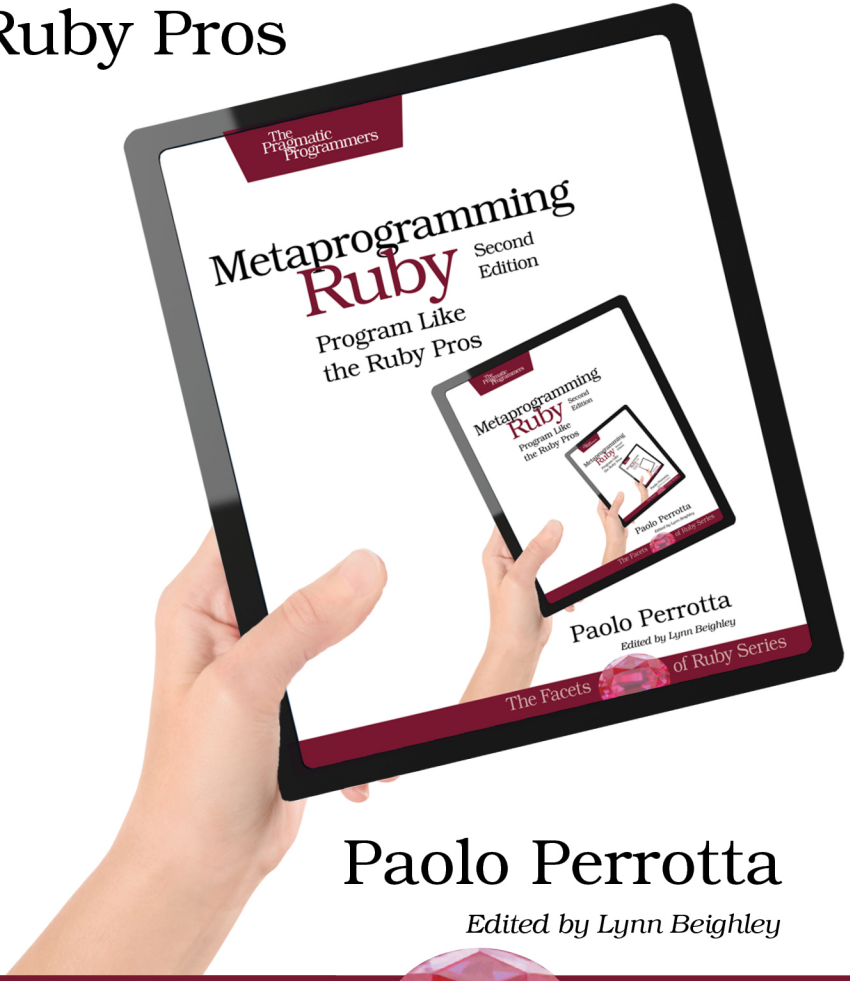
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Metaprogramming Ruby

Second Edition

Program Like the Ruby Pros



Paolo Perrotta

Edited by Lynn Beighley

The Facets of Ruby Series

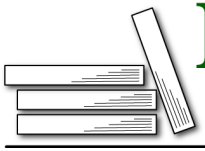
Metaprogramming Ruby 2

Program Like the Ruby Pros

Paolo Perrotta

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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://pragprog.com*.

The team that produced this book includes:

Lynn Beighley (editor)
Potomac Indexing, LLC (indexer)
Cathleen Small (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-94122-212-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2014

I was thirteen, and I was tired of hanging out at the local toy shop to play Intellivision games. I wanted my own videogame console. I'd been bugging my parents for a while, with no success.

Then I found an alternative: I could play games on a computer as well. So I asked my parents to buy me one of those new 8-bit computers—you know, to learn useful stuff. My dad agreed, and my mom took me to the shop and bought me a Sinclair ZX Spectrum.

Mom, Dad… Here is something that I should've told you more often in my life: thank you. This book is dedicated to the two of you. I'm hoping it will make you proud, just like your once-kid is proud of you. And while I'm here, I have something to confess about that life-changing day thirty years ago: I didn't really want to learn stuff. I just wanted to play.

In fact, that's what I've been doing all these years.

Blocks Are Closures

Where you find there is more to blocks than meets the eye and you learn how to smuggle variables across scopes.

As Bill notes on a piece of scratch paper, a block is not just a floating piece of code. You can't run code in a vacuum. When code runs, it needs an *environment*: local variables, instance variables, self....

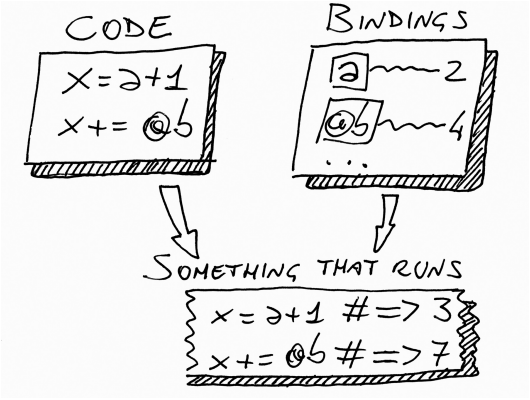


Figure 6—Code that runs is actually made up of two things: the code itself and a set of bindings.

Because these entities are basically names bound to objects, you can call them the *bindings* for short. The main point about blocks is that they are all inclusive and come ready to run. They contain both the code *and* a set of bindings.

You're probably wondering where the block picks up its bindings. When you define the block, it simply grabs the bindings that are there at that moment, and then it carries those bindings along when you pass the block into a method:

blocks/blocks_and_bindings.rb

```
def my_method
  x = "Goodbye"
  yield("cruel")
end

x = "Hello"
my_method {|y| "#{x}, #{y} world" } # => "Hello, cruel world"
```

When you create the block, you capture the local bindings, such as x. Then you pass the block to a method that has its own separate set of bindings. In the previous example, those bindings also include a variable named x. Still, the code in the block sees the x that was around when the block was defined, not the method's x, which is not visible at all in the block.

You can also define additional bindings inside the block, but they disappear after the block ends:

blocks/block_local_vars_failure.rb

```
def just_yield
  yield
end

top_level_variable = 1

just_yield do
  top_level_variable += 1
  local_to_block = 1
end

top_level_variable  # => 2
local_to_block      # => Error!
```

Because of the properties above, a computer scientist would say that a block is a *closure*. For the rest of us, this means a block captures the local bindings and carries them along with it.

So, how do you use closures in practice? To understand that, take a closer look at the place where all the bindings reside—the *scope*. Here you'll learn to identify the spots where a program changes scope, and you'll encounter a particular problem with changing scopes that can be solved with closures.

Scope

Imagine being a little debugger making your way through a Ruby program. You jump from statement to statement until you finally hit a breakpoint. Now catch your breath and look around. See the scenery around you? That's your *scope*.

You can see bindings all over the scope. Look down at your feet, and you see a bunch of local variables. Raise your head, and you see that you're standing within an object, with its own methods and instance variables; that's the current object, also known as self. Farther away, you see the tree of constants so clear that you could mark your current position on a map. Squint your eyes, and you can even see a bunch of global variables off in the distance.

But what happens when you get tired of the scenery and decide to move on?

Changing Scope

This example shows how scope changes as your program runs, tracking the names of bindings with the Kernel#local_variables method:

blocks/scopes.rb

```
v1 = 1
class MyClass
  v2 = 2
  local_variables    # => [:v2]
  def my_method
    v3 = 3
    local_variables
  end
  local_variables    # => [:v2]
end

obj = MyClass.new
obj.my_method        # => [:v3]
obj.my_method        # => [:v3]
local_variables      # => [:v1, :obj]
```

Track the program as it moves through scopes. It starts within the top-level scope that you read about in *The Top Level*, on page ?. After defining v1 in the top-level scope, the program enters the scope of MyClass's definition. What happens then?

Some languages, such as Java and C#, allow "inner scopes" to see variables from "outer scopes." That kind of nested visibility doesn't happen in Ruby, where scopes are sharply separated: as soon as you enter a new scope, the previous bindings are replaced by a new set of bindings. This means that when the program enters MyClass, v1 "falls out of scope" and is no longer visible.

In the scope of the definition of MyClass, the program defines v2 and a method. The code in the method isn't executed yet, so the program never opens a new scope until the end of the class definition. There, the scope opened with the class keyword is closed, and the program gets back to the top-level scope.

What happens when the program creates a MyClass object and calls my_method twice? The first time the program enters my_method, it opens a new scope and defines a local variable, v3. Then the program exits the method, falling back to the top-level scope. At this point, the method's scope is lost. When the program calls my_method a second time, it opens yet another new scope, and it defines a new v3 variable (unrelated to the previous v3, which is now lost).

Global Variables and Top-Level Instance Variables

Global variables can be accessed by any scope:

```
def a_scope
  $var = "some value"
end

def another_scope
  $var
end

a_scope
another_scope # => "some value"
```

The problem with global variables is that every part of the system can change them, so in no time you'll find it difficult to track who is changing what. For this reason, the general rule is this: when it comes to global variables, use them sparingly, if ever.

You can sometimes use a top-level instance variable in place of a global variable. These are the instance variables of the top-level main object, described in *The Top Level*, on page ?:

```
@var = "The top-level @var"

def my_method
  @var
end

my_method # => "The top-level @var"
```

You can access a top-level instance variable whenever main takes the role of self, as in the previous example. When any other object is self, the top-level instance variable is out of scope.

```
class MyClass
  def my_method
    @var = "This is not the top-level @var!"
  end
end
```

Being less universally accessible, top-level instance variables are generally considered safer than global variables—but not by a wide margin.

Finally, the program returns to the top-level scope, where you can see v1 and obj again. Phew!

Here is the example's important point: "Whenever the program changes scope, some bindings are replaced by a new set of bindings." Granted, this doesn't happen to all the bindings each and every time. For example, if a method calls another method on the same object, instance variables stay in scope through the call. In general, though, bindings tend to fall out of scope when

the scope changes. In particular, local variables change at every new scope. (That's why they're "local.")

As you can see, keeping track of scopes can be a tricky task. You can spot scopes more quickly if you learn about *Scope Gates*.

Scope Gates

There are exactly three places where a program leaves the previous scope behind and opens a new one:

- Class definitions
- Module definitions
- Methods

Scope changes whenever the program enters (or exits) a class or module definition or a method. These three borders are marked by the keywords `class`, `module`, and `def`, respectively. Each of these keywords acts like a *Scope Gate*.

Spell: Scope Gate

For example, here is the previous example program again, with Scope Gates clearly marked by comments:

```
v1 = 1
class MyClass        # SCOPE GATE: entering class
  v2 = 2
  local_variables    # => ["v2"]
  def my_method      # SCOPE GATE: entering def
    v3 = 3
    local_variables
  end                # SCOPE GATE: leaving def
  local_variables    # => ["v2"]
end                  # SCOPE GATE: leaving class

obj = MyClass.new
obj.my_method        # => [:v3]
local_variables      # => [:v1, :obj]
```

Now it's easy to see that this program opens three separate scopes: the top-level scope, one new scope when it enters `MyClass`, and one new scope when it calls `my_method`.

There is a subtle difference between `class` and `module` on one side and `def` on the other. The code in a class or module definition is executed immediately. Conversely, the code in a method definition is executed later, when you eventually call the method. However, as you write your program, you usually don't care *when* it changes scope—you only care that it does.

Now you can pinpoint the places where your program changes scope—the spots marked by class, module, and def. But what if you want to pass a variable through one of these spots? This question takes you back to blocks.

Flattening the Scope

The more you become proficient in Ruby, the more you get into difficult situations where you want to pass bindings through a *Scope Gate (11)*:

blocks/flat_scope_1.rb
```
my_var = "Success"

class MyClass
  # We want to print my_var here...
  def my_method
    #  ..and here
  end
end
```

Scope Gates are quite a formidable barrier. As soon as you walk through one of them, local variables fall out of scope. So, how can you carry my_var across not one but two Scope Gates?

Look at the class Scope Gate first. You can't pass my_var through it, but you can replace class with something else that is not a Scope Gate: a method call. If you could call a method instead of using the class keyword, you could capture my_var in a closure and pass that closure to the method. Can you think of a method that does the same thing that class does?

If you look at Ruby's documentation, you'll find the answer: Class.new is a perfect replacement for class. You can also define instance methods in the class if you pass a block to Class.new:

blocks/flat_scope_2.rb
```
my_var = "Success"

MyClass = Class.new do
  # Now we can print my_var here...
  puts "#{my_var} in the class definition!"

  def my_method
    # ...but how can we print it here?
  end
end
```

Now, how can you pass my_var through the def Scope Gate? Once again, you have to replace the keyword with a method call. Think of the discussion about *Dynamic Methods (?)*: instead of def, you can use Module#define_method:

blocks/flat_scope_3.rb

```
my_var = "Success"

MyClass = Class.new do
  "#{my_var} in the class definition"

  define_method :my_method do
    "#{my_var} in the method"
  end
end

MyClass.new.my_method

require_relative "../test/assertions"
assert_equals "Success in the method", MyClass.new.my_method
```

```
Success in the class definition
Success in the method
```

If you replace Scope Gates with method calls, you allow one scope to see variables from another scope. Technically, this trick should be called *nested lexical scopes*, but many Ruby coders refer to it simply as "flattening the scope," meaning that the two scopes share variables as if the scopes were squeezed together. For short, you can call this spell a *Flat Scope*.

Spell: Flat Scope

Sharing the Scope

Once you know about *Flat Scopes (13)*, you can do pretty much whatever you want with scopes. For example, assume that you want to share a variable among a few methods, and you don't want anybody else to see that variable. You can do that by defining all the methods in the same Flat Scope as the variable:

blocks/shared_scope.rb

```
def define_methods
  shared = 0

  Kernel.send :define_method, :counter do
    shared
  end

  Kernel.send :define_method, :inc do |x|
    shared += x
  end
end

define_methods

counter       # => 0
inc(4)
```

```
counter       # => 4
```

This example defines two *Kernel Methods (?)*. (It also uses *Dynamic Dispatch (?)* to access the private class method define_method on Kernel.) Both Kernel#counter and Kernel#inc can see the shared variable. No other method can see shared, because it's protected by a *Scope Gate (11)*—that's what the define_methods method is for. This smart way to control the sharing of variables is called a *Shared Scope*.

Spell: Shared Scope

Shared Scopes are not used much in practice, but they're a powerful trick and a good example of the power of scopes. With a combination of Scope Gates, Flat Scopes, and Shared Scopes, you can twist and bend your scopes to see exactly the variables you need, from the place you want. Now that you wield this power, it's time for a wrap-up of Ruby closures.

Closures Wrap-Up

Each Ruby scope contains a bunch of bindings, and the scopes are separated by *Scope Gates (11)*: class, module, and def.

If you want to sneak a binding or two through a Scope Gate, you can use blocks. A block is a *closure*: when you define a block, it captures the bindings in the current environment and carries them around. So you can replace the Scope Gate with a method call, capture the current bindings in a closure, and pass the closure to the method.

You can replace class with Class.new, module with Module.new, and def with Module#define_method. This is a *Flat Scope (13)*, the basic closure-related spell.

If you define multiple methods in the same Flat Scope, maybe protected by a Scope Gate, all those methods can share bindings. That's called a *Shared Scope (14)*.

Bill glances at the road map he created. (See *Today's Roadmap*, on page ?.) "Now that you've gotten a taste of Flat Scopes, we should move on to something more advanced: instance_eval."