This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 34 Star 123 Fork 411

Code Issues 4 Pull requests 12 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / ruby / readings / references.md Find file Copy path

loschorts reverse directory structure to go topics -> types c1e6862 on May 16, 2016

1 contributor

150 lines (115 sloc) 3.6 KB Raw Blame History

References

In the Ruby language variables hold references to objects.

```
my_object = Object.new
# my_object now refers to a new instance of Object

my_object = Object.new
# a new Object instance is created, and the my_object variable is
# reassigned so that it now refers to this object, rather than the old
# object.

# call `my_method` on the object that `my_object` refers to
my_object.my_method
```

One point is that `=` does not mutate (modify) an object; it merely reassigns the variable so that it now refers to a new object.

Here is another example.

```
def add_to_array!(array, item)
  array << item
end

my_array = []
add_to_array!(my_array, "an item!")
p my_array
```

This code will modify the array referred to by `my_array`. When we call `add_to_array`, we're passing it a reference to the same object referred to by `my_array`, as well as a reference to the literal string `"an item!"`. So inside `add_to_array`, the variable `array` refers to the same object that `my_array` refers to outside the method. So modifications to the array inside the method will be visible outside.

The `[]=` and accessor methods

We've said that `=` never modifies an object, and merely reassigns a variable to refer to a new object. But then how do I explain this:

```
my_hash = {}
# add a key & value to the hash
my_hash[:key] = :value
```

How does this work? Notice that we aren't just assigning to a variable; we're asking to mutate `my_hash` so that `:key` will refer to `:value` now. To do this, Ruby calls the `[]=` method on the hash. Here's how it might be defined:

```
class Hash
  def []=(key, val)
    # code to set key so that it maps to value...
  end
end
```

In fact, we can even write this with the usual method call notation:

```
my_hash.[]=(:key, :value)
```

Other writer methods work similarly:

```
class Cat
  attr_accessor :name
end

my_cat = Cat.new
my_cat.name = "Breakfast" # == my_cat.name=("Breakfast")
```

Conveniences like this (the ability to write `hash[:key] = :value` instead of `hash.[]=(:key, :value)`) are called *syntactic sugar*. They don't change the fundamental way the language works; they are merely shortcuts to make things a little easier on us.

`+=` and `!=`

One last bit of syntactic sugar is to discuss how Ruby interprets `+=` and `!=`. This is done quite simply

```
x += y # => x = x + y, will work as long as `#+` is defined
x != y # => !(x == y), will use the `#==` method
```

So while `[]=` was a method, `+=` and `!=` are pure syntactic sugar. Ruby will translate them into code that calls `+` and `==` respectively.

The or trick

The Ruby `||` operation does something called short circuiting:

```
true || this_code_is_not_run
false || this_code_will_be_run
```

Ruby is sort of lazy; it won't evaluate the right side if the left side of `||` is already true. That makes sense, because regardless of what the right side is, the whole or statement will always be true. Ruby understands this, and so as a special rule won't execute more than it needs to.

Ruby will return the first "truthy" value from the or:

```
(1 || 2) == 1
(nil || 5) == 5
```

There's an "or trick" that uses `||=`:

```ruby
class MemoizedFibonacci
  def initialize
    @values = {}
  end

  def fib(n)
    @values[n] ||= calculate_fib(n)
    # expands to:
    #     @values[n] = @values[n] || calculate_fib(n)
    # if @values[n] is nil (not previously computed), will call
    # `calculate_fib` and store the result for later reference.
  end

  private
  def calculate_fib(n)
    case n
    when 0
      0
    when 1
      1
    else
      fib(n - 1) + fib(n - 2)
    end
  end
end
```

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 34 Star 123 Fork 411

Code Issues 4 Pull requests 12 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / ruby / readings / array-hash-defaults.md Find file Copy path

loschorts reverse directory structure to go topics -> types c1e6862 on May 16, 2016

1 contributor

197 lines (154 sloc) 5.54 KB Raw Blame History

Array/Hash Defaults

Arrays of Arrays

Here's a very common problem that everyone will run into. Let's say we want to make an array of arrays:

```
[4] pry(main)> arr_of_arrs = Array.new(3, [])
=> [[], [], []]
[5] pry(main)> arr_of_arrs[0] << "a"
=> ["a"]
[6] pry(main)> arr_of_arrs
=> [["a"], ["a"], ["a"]]
```

Wait, what happened? We added `"a"` to the first array, but all of them were modified?

The reason is that only two arrays are created in the example: (1) `arr_of_arrs`, (2) the single empty array passed into the `Array` constructor (`[]`). `arr_of_arrs` stores three references to the same empty array.

Thus, when you access the array at position `0` and mutate it, you're mutating the same array referenced by position at one and two.

The way to solve this problem is like so:

```
[7] pry(main)> arr_of_arrs = Array.new(3) { Array.new }
=> [[], [], []]
[8] pry(main)> arr_of_arrs[0] << "a"
=> ["a"]
[9] pry(main)> arr_of_arrs
=> [["a"], [], []]
```

Here, instead of passing a reference to a single empty array (which would be stored at three locations), we've passed a block. The block will be run to produce a value to store for each position in the array. The block constructs a new array each time it is run.

I wrote `Array.new` in the block to make it clear that a new array is constructed each time the block is executed, but you could equivalently write `[]` in the block.

Mutable vs Immutable

Okay, we've seen that arrays store references to objects. We've seen a naive problem where we stored several references to the same object. Since all references refer to the same underlying object, a mutation through one reference (`arr_of_arrs[0] << "a"`) is also visible through another reference (`arr_of_arrs[1] == ["a"]`).

You may have previously written code like this:

```
arr2 = Array.new(3, 1)

arr2[0] += 1
arr2[0] == 2
arr2[1] == 1
arr2[2] == 1
```

Does this contradict what we've just discussed about references and mutations? Why isn't the change visible at position `1` ?

Let's unpack `arr2[0] += 1` . This is Ruby shorthand for:

```
arr2[0] = arr2[0] + 1
```

Let's further break this into steps:

0. First, fetch the number at position 0 (which is `1`).
1. Next, add one to this number. This creates a new number object. The `+` operation does not mutate the original object.
2. Finally, assign a reference to the new object (`2`) to position `0` of `arr` .

The trick is that we never mutate any number. We produce a new one and reset `arr2[0]` to refer to the new object. That's why none of the other indices are affected.

The `Fixnum` and `Float` classes are called immutable. None of their methods modify data inside the `Fixnum` / `Float` ; they produce new values instead. `nil` is another example.

Hash default values

Providing a default value for a Hash has the same issues as with an Array:

```
[1] pry(main)> cats = Hash.new([])
=> {}
[2] pry(main)> cats["Devon"]
=> []
[3] pry(main)> cats
=> {}
```

Providing an argument to `Hash.new` merely changes what is returned when we look up a key that isn't present in the hash. Note how this doesn't assign a value to `"Devon"` through mere access of the key. To do that, we can do something like:

```
[4] pry(main)> cats["Devon"] += ["Earl"]
=> ["Earl"]
[5] pry(main)> cats
=> {"Devon"=>["Earl"]}
[6] pry(main)> cats["Devon"] += ["Breakfast"]
=> ["Earl", "Breakfast"]
[7] pry(main)> cats
=> {"Devon"=>["Earl", "Breakfast"]}
```

Better. `cats["Devon"] += ["Earl"]` means `cats["Devon"] = cats["Devon"] + ["Earl"]`. This constructs a new array and stores it for key `"Devon"`

But what about this?

```
[7] pry(main)> cats = Hash.new([])
=> {}
[8] pry(main)> cats["John"] << "Kiki"
=> ["Kiki"]
[9] pry(main)> cats
=> {}
[10] pry(main)> cats["Raul"]
=> ["Kiki"]
```

Let's think through what's happening here. On line 8, we try to get a value for `cats["John"]`. `"John"` is not a key in the hash, so the default (an empty array) is returned. We then mutate the default value by adding `"Kiki"` to it.

We never set a value for `"John"` though, so this is not stored in the Hash (see the result of line 9).

Later, when we try to access some other non-present key (`"Raul"`), the default value is returned again. But since we mutated the value by shovelling `"Kiki"` in, this is no longer empty. This is bad, because we never meant for `"Raul"` to own `"Kiki"`.

We can start to fix the problem as before:

```
[11] pry(main)> cats2 = Hash.new() { [] }
=> {}
[12] pry(main)> cats2["Devon"] << "Breakfast"
=> ["Breakfast"]
[13] pry(main)> cats2["George"]
=> []
```

Hash will use the block to produce a new default value each time. Modifying the value won't have an affect on looking up other non-existent keys, since we create a new value each time, instead of reusing a single default object.

But we have the other problem again: we're still not setting a value.

```
[16] pry(main)> cats2
=> {}
```

Let's fix this:

```
[17] pry(main)> cats3 = Hash.new { |h, k| h[k] = [] }
=> {}
[18] pry(main)> cats3["Devon"]
=> []
[19] pry(main)> cats3
=> {"Devon"=>[]}
[20] pry(main)> cats3["John"] << "Kiki"
=> ["Kiki"]
[21] pry(main)> cats3
=> {"Devon"=>[], "John"=>["Kiki"]}
```

Here we've modified the block to take two arguments: when Hash needs a default value, it will pass itself (`h`) and the key (`k`). The block will not only create an empty array, but also assign it to the hash.

You can see one somewhat funny side-effect when we look up `"Devon"`; even when we just want to lookup a value, if it is not present we'll incur the side-effect of mutating the hash (the key `"Devon"` got added).

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 34 Star 123 Fork 411

Code Issues 4 Pull requests 12 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / ruby / readings / scope.md Find file Copy path

loschorts reverse directory structure to go topics -> types c1e6862 on May 16, 2016

1 contributor

79 lines (57 sloc) 1.8 KB Raw Blame History

Scope

Scope is the context in which a variable name is valid and can be used.

A name is in scope (accessible) if the name has been previously defined in the current method (called a local variable) or at a higher level of the current method. A new level starts whenever we begin a class, a method, or a block.

We can't use a variable before it is defined:

```ruby
def pow(base, exponent)
  i = 0
  while i < exponent
    result = result * base  # Error: result is being used before it has been defined.

    i += 1
  end

  result
end
```

We can't use a variable from a deeper scope. In the below example, `cat_age` is defined inside `extract_cat_age` and not available at the top-level scope.

```ruby
# defines a cat variable name in the global scope
cat = {
  :name => "Breakfast",
  :age => 8
}

def extract_cat_age
  # creates a new local variable inside this method; won't add
  # variable to global scope; variable will be lost when method is
  # over

  cat_age = cat[:age]
end

extract_cat_age
cat_age # ERROR: variable out of scope
```

Sometimes things are subtle:

```
def fourth_power(i)
  square(i) * square(i)   # wait, isn't square not defined yet?
end

def square(i)
  i * i
end

# Ah, but by the time we _call_ `fourth_power` and run the
# interior code, `square` will have been defined.

fourth_power(2)
```

Global variables

NOTE: This last bit about global variables is not essential.

While you shouldn't typically create global variables, you can do so with the `$` prefix.

You can also create file-local global variables by defining a variable without `$` at the top-level (outside any block, method, class). However, file-local global variable won't be accessible to other Ruby files that load the file.

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 34 Star 123 Fork 411

Code Issues 4 Pull requests 12 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / ruby / readings / refactoring.md Find file Copy path

loschorts fix broken links in ruby/readings/refactoring.md (#33) cea99e4 on May 26, 2016

1 contributor

128 lines (98 sloc) 5.44 KB Raw Blame History

Refactoring (code smells)

From the Wiki:

> A code smell is any symptom in the source code of a program that possibly indicates a deeper problem. Code smells are usually not bugs — they are not technically incorrect and don't currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future ... a code smell is a driver for refactoring.

We list a few kinds of smells here here:

Duplicated/similar code: see the reading on DRY code.

Long methods: see the reading on method decomposition.

Too many parameters: The more parameters a method has, the greater the chance that it is too coupled to code that invokes it. Many parameters may also be a sign of an excessively complex method.

Data clump: if you see a group of data always being passed around together, this is usually a good candidate for refactoring out into an object. For example `start_date` and `end_date` might be pulled out into a `date_range` object.

Long method chains: long method chains can often be a sign that you're violating the Law of Demeter: only talk to your "neighbors", only use one dot.

```
bicycle.front_tire.rotate
bicycle.rear_tire.rotate

# vs
bicycle.rotate_tires
```

The idea is that instead of reaching into objects (like the `bicycle`) and calling methods on their "internals" (`front_tire` and `rear_tire`), we should create a method that will take care of this for us (`rotate_tires`). This helps us organize our code, and prevents us from becoming too dependent on the internal structure of other objects.

LoD has disadvantages (see the wiki article); if taken too literally you end up with overly wide interfaces. However, the longer your method chains get, the more likely you should apply LoD.

Indecent Exposure: Classes should share the bare minimum interface with the outside world. If you don't have a compelling reason to make a method or variable public, hide it. Other classes will interact with the public surface of the class; any change to the public surface will necessitate changes to interacting classes. Minimizing exposure will better enable change.

From the wiki page: "Coupling (dependency) is the degree to which each program module relies on each one of the other modules."

Indecent exposure may result in coupling that is too tight. The more extensive your classes' interface, the more tightly other classes can latch on to it.

A complicated interface can also signal a class that is doing too much. Probably there's an opportunity to break out responsibilities to other classes.

We want to minimize the amount of coupling between our classes and objects. To understand how tightly coupled your classes are, ask yourself if you changed the internals of one class, would you have to change things in the other? If you do, you've likely coupled the two classes too tightly.

Speculative Generality: Follow the principle of YAGNI ('You ain't gonna need it'). New devs often waste time thinking of all the ways they might "generalize" their code so that it's easy to make this change or that change, so that they can support all sorts of new features easily, etc., etc. This can result in a very complicated design, before any business need for those features even exists.

More importantly, it is very, very hard to get a speculative design like this right. Good design happens when there are clear requirements and concrete examples to think about; it's hard to make a perfect design for some hazy, poorly understood future. Don't solve *abstract* problems: wait until you have a *concrete* problem.

God object: A god object is one that is very tightly connected to all the other objects in the system. Good OO design results in classes that are lightly coupled. A good class delegates responsibility as necessary to other objects; it shouldn't need to know everything about what every other object is doing (omniscience), and it shouldn't micromanage how other objects manage their responsibilities. Nothing in your program usually needs to even know about the existence of everything else.

Dead code: don't leave commented-out (or otherwise unused) code in your code base. This is why we have version control (we'll learn about git soon!). Clean up after yourself!

Clean Code

When push comes to shove, sometimes you need to make compromises; when deadlines hit, we all ship code we aren't 100% proud of. Still, endeavor to write clean code. Pay attention to style. Pay attention to code organization. Pay attention to code smells. Refactor often and aggressively. Bask in the beauty of clean code. Take pride in it.

Further Reading

- Enjoy watching Ben Orenstein's classic talk: Refactoring: From Good to Great.

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About