**Class Instance Variables**

You know all about instance variables:

class Dog

def initialize(name)

@name = name

end

# could also use `attr\_reader :name` to generate this.

def name

@name

end

end

Inside a method, we can set an instance variable of the current object. This is what we do inside the initialize instance method.

Recall that classes are objects, too. For instance, Dog itself is a class. We can set instance variables on the Dog class object too:

class Dog

def self.all

@dogs ||= []

end

def initialize(name)

@name = name

self.class.all << self

end

end

In the class method all, we fetch/assign an instance variable dogs. This stores an instance variable in the Dog object. As part of the initialization of a Dog instance, we add the Dog instance to the list of all Dogs. We can access all dogs through Dog.all:

d1 = Dog.new("Fido")

d2 = Dog.new("Fido 2.0")

p Dog.all

=> [#<Dog:0x007fe140a23928 @name="Fido">,

#<Dog:0x007fe140a628d0 @name="Fido 2.0">]

Note that the @dogs variable in Dog.all works the same as any other instance variable: setting or accessing @dogs will look inside the current object (in this case, the Dog class object) and set/fetch the instance variable.

When an instance variable is stored on a class, it is sometimes called a **class instance variable**. Don't let the name wow you though; we're just using a typical instance variable. This is similar to how class methods are merely methods that are called on a Class object.

**Inheritance:**@@

For our purposes, the standard instance variable will typically be enough. There is one downside: class instance variables don't interact very nicely with inheritance. Let's take an example:

class Corgi < Dog

end

Let's think what happens when we run Corgi.new("Linus"). Per the definition of initialize in Dog, we will run self.class.all << self. self.class is Corgi; Corgi will have an all method by virtue of inheriting from Dog.

The all method will look in Corgi for a @dogs instance variable. Note that Corgi will not share the @dogs variable from Dog. Corgi and Dog are different objects, so they do not share instance variables. This means that Corgi will have its own @dogs variable, and Corgis will not be added to the Dog's array of @dogs.

That may not be what you want. Perhaps you would like that Corgis be added to the list of all Dogs. You can do this by switching from @dogs to @@dogs; @@dogs is a **class variable**.

Class variables (not class **instance** variables) are shared between super-class and subclass. Let's see this:

class Dog

def self.all

@@dogs ||= []

end

def initialize(name)

@name = name

self.class.all << self

end

end

class Husky < Dog

end

h = Husky.new("Rex")

Dog.all # => #<Husky:0x007f95421b5560 @name="Rex">

I should note: most of the classes you write won't be inherited from. So you may want to eject the emotional baggage of @@ and just stick with the @ variables you are familiar with, at least until @ doesn't work.

**Global variables**

Bonus topic: buy two get one free! Global variables!

Global variables are prefixed with a $. Global variables are top-level variables that live outside any class. They are accessible anywhere:

# this should have been a class variable though...

$all\_dogs = []

class Dog

def self.all

$all\_dogs

end

def initialize(name)

@name = name

$all\_dogs << self

end

end

Why couldn't we write all\_dogs = []? The reason is that if we try to create a **local** variable at the top level scope, it will be cleaned up and removed when the source file is executed:

[1] pry(main)> require './dog'

=> true

[2] pry(main)> Dog.all

NameError: undefined local variable or method `all\_dogs' for Dog:Class

from: /Users/ruggeri/test.rb:5:in `all'

from: (pry):2:in `\_\_pry\_\_'

Global variables, on the other hand, have permanence.

**Avoid global variables**

Global variables are not very common to use; you should avoid them. I rarely if ever use them. The reason is that since global variables live outside any class, they aren't very object oriented. Data is normally stored in one of two places:

* Inside an object (instance, class instance, and class variables)
* Inside a local variable; the local variable lives as long as the current method call.

If you need to access an object inside a method, it is typical to pass the object into the method. If you need to return a result from a method, it is typical to use return to pass it back. There is seldom a reason to store things globally.

There are occasionally exceptions: sometimes an object will be useful throughout your entire program, in which case you may want to make it globally accessible. One classic example is the $stdin and $stdout variables, which contain File objects (technically, IO objects, but they're very similar) that you can use to read/write to the user.

Here's how puts and gets are defined:

def puts(\*args)

$stdout.puts(\*args)

end

def gets(\*args)

$stdin.gets(\*args)

end

This eliminates most of the need to use these variables explicitly. However, say you wanted to write your output differently depending on whether the user was reading your output in a terminal or dumping your output to a file. In Bash, they can specify this by either:

$ ruby program.rb # print to console

$ ruby program.rb > ./file\_to\_print\_to # print to a file

You could use the IO#isatty method of $stdout to do this:

if $stdout.isatty

puts "I'm on a console!"

else

puts "I'm on a file!"

end

Great. However, you're not likely to need to use global variables your own self.

module Wet

class Model

def save

# common code to save a model to a DB

end

end

class Cat < Model

def save

super # call Model#save implementation

puts "Cat saved!" # perform Cat-specific post-save logic

end

end

class Dog < Model

def save

super # call Model#save implementation

puts "Dog saved!" # perform Dog-specific post-save logic

end

end

end

module Dry

class Model

def self.save\_callbacks

@save\_callbacks ||= []

end

# define a macro to \*register\* (record) a post-save \*callback\* (a

# method to be called after another method completes)

def self.save\_callback(method\_name)

self.save\_callbacks << method\_name

end

def save

# common code to save a model to a DB

run\_save\_callbacks

end

private

def run\_save\_callbacks

self.class.save\_callbacks.each do |method\_name|

# invoke the method named `method\_name`

send(method\_name)

end

end

end

class Cat < Model

save\_callback :print\_cat\_saved

private

def print\_cat\_saved

puts "Cat saved!" # perform Cat-specific post-save logic

end

end

class Dog < Model

save\_callback :print\_dog\_saved

private

def print\_dog\_saved

puts "Dog saved!" # perform Dog-specific post-save logic

end

end

end

# Notice the redundant implementations of moo/bark/meow.

module Wet

class Animal

end

class Cow < Animal

def moo

puts "moo!"

end

end

class Dog < Animal

def bark

puts "bark!"

end

end

class Cat < Animal

def meow

puts "meow!"

end

end

end

# Let's DRY it up!

module Dry

class Animal

private

# define a `makes\_noise` class method; a method that defines other

# methods is called a \*macro\*

def self.makes\_noise(name)

# When an Animal subclass calls `makes\_noise`, we will

# \*dynamically\* define a new method on the

# subclass. `Class#define\_method` is an instance method of the

# `Class` class.

define\_method(name) do

puts "#{name}!"

end

nil

end

end

class Cow < Animal

makes\_noise :moo

end

class Dog < Animal

makes\_noise :bark

end

class Cat < Animal

makes\_noise :meow

end

end