**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 ||=:

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