LaunchSchool
A School for Software Engineers

Blog

- RSS

SEARCH

Mutating and Non-Mutating Methods in Ruby

This is the second in a series of three articles that discuss how Ruby manipulates variables and objects, and, in particular, how objects are passed around in a Ruby program. In the Understand Variable References and Mutability article, we explored how Ruby uses variables — variables don't actually contain values, but instead serve as references to objects. We also discussed the concepts of object mutability and immutability, and introduced the concepts of pass by value and pass by reference.

In this article, we discuss methods, and how they can be mutating or non-mutating with respect to certain arguments. We focus special attention on assignment and concatenation, two operations that cause a lot of confusion for new rubyists.

Mutating and Non-Mutating Methods

Methods can be either mutating or non-mutating. As you might expect, mutating methods change something; non-mutating methods do not. The object that may or may not be mutated is of concern when discussing whether a method is mutating or non-mutating. For example, the method `String#sub!` is mutating with respect to the String used to call it, but non-mutating with respect to its arguments.

Non-Mutating Methods

A method is said to be non-mutating with respect to an argument or its calling object if it does not modify it. Most methods you will encounter do not mutate their arguments or caller. Some do mutate their caller, but few mutate the arguments.

All methods are non-mutating with respect to immutable objects. A method simply can't modify an immutable object. Thus, any method that operates on numbers and boolean values is guaranteed to be non-mutating with respect to that value.

Assignment is Non-Mutating

Of particular interest when discussing non-mutating methods is assignment with =. As we saw in Variable References and Mutability article, assignment merely tells Ruby to bind an object to a variable. This means that assignment does not change an object; it merely connects the variable to a new object. While = is not an actual method in Ruby, it acts like a non-mutating method, and should be treated as such.

Take a moment to study this code:

```
1 def fix(value)
2   value.upcase!
3   value.concat('!')
4   value
5 end
6
7 s = 'hello'
8 t = fix(s)
```

When this code runs, what values do `s` and `t` have?

We start by passing `s` to `fix`; this binds the String represented by "hello" to `value`. In addition, `s` and `value` are now aliases for the String.

Next, we call `#upcase!` which converts the String to uppercase. A new String is not created; the String that is referenced by both `s` and `value` now contains the value `"HELLO"`.

We then call `#concat` on `value`, which also modifies `value` instead of creating a new String; the String now has a value of `"HELLO!"`, and both `s` and `value` reference that object.

Finally, we return a reference to the String and store it in `t`.

The only place we create a new String in this code is when we assign "hello" to `s`. The rest of the time, we work directly with that object, modifying it as needed. Thus, both `s` and `t` reference the same String, and that String has the value `'HELLO!'`. You can verify this yourself by running this code in irb:

```
1  >> def fix(value)
2  --   value.upcase!
3  --   value.concat('!')
4  -- end
5  => :fix
6
7  >> s = 'hello'
8  => "hello"
9
10 >> s.object_id
11 => 70363946430440
12
13 >> t = fix(s)
14 => "HELLO!"
15
16 >> s
17 => "HELLO!"
18
19 >> t
20 => "HELLO!"
21
22 >> s.object_id
23 => 70363946430440
24
25 >> t.object_id
26 => 70363946430440
```

Let's modify the original code slightly:

```
1 def fix(value)
2   value = value.upcase
3   value.concat('!')
4 end
5
```

```
6 s = 'hello'
7 t = fix(s)
```

Now what happens with `s` and `t`?

In this modified code, we assign the return value of `value.upcase` back to `value`. Unlike `#upcase!`, `#upcase` does not modify the String referenced by `value`; instead, it creates a new copy of the String referenced by `value`, modifies the copy, and then returns a reference to the copy. We then bind `value` to the returned reference.

The rest of the program runs as before, but if you look at the results in irb, you'll see that things are quite different:

```
1  >> def fix(value)
2  --   value = value.upcase
3  --   value.concat('!')
4  -- end
5  => :fix
6
7  >> s = 'hello'
8  => "hello"
9
10 >> s.object_id
11 => 70349169469400
12
13 >> t = fix(s)
14 => "HELLO!"
15
16 >> s
17 => "hello"
18
19 >> t
20 => "HELLO!"
21
22 >> s.object_id
23 => 70349169469400
24
25 >> t.object_id
26 => 70349169435840
```

`s` and `t` now reference different objects, and the String referenced by `s` has not been modified. What happened here?

Let's modify our code again:

```
1  def fix(value)
2    puts "initial object #{value.object_id}"
3    value = value.upcase
4    puts "upcased object #{value.object_id}"
5    value.concat('!')
6  end
7
8  s = 'hello'
9  puts "original object #{s.object_id}"
10 t = fix(s)
11 puts "final object #{t.object_id}"
```

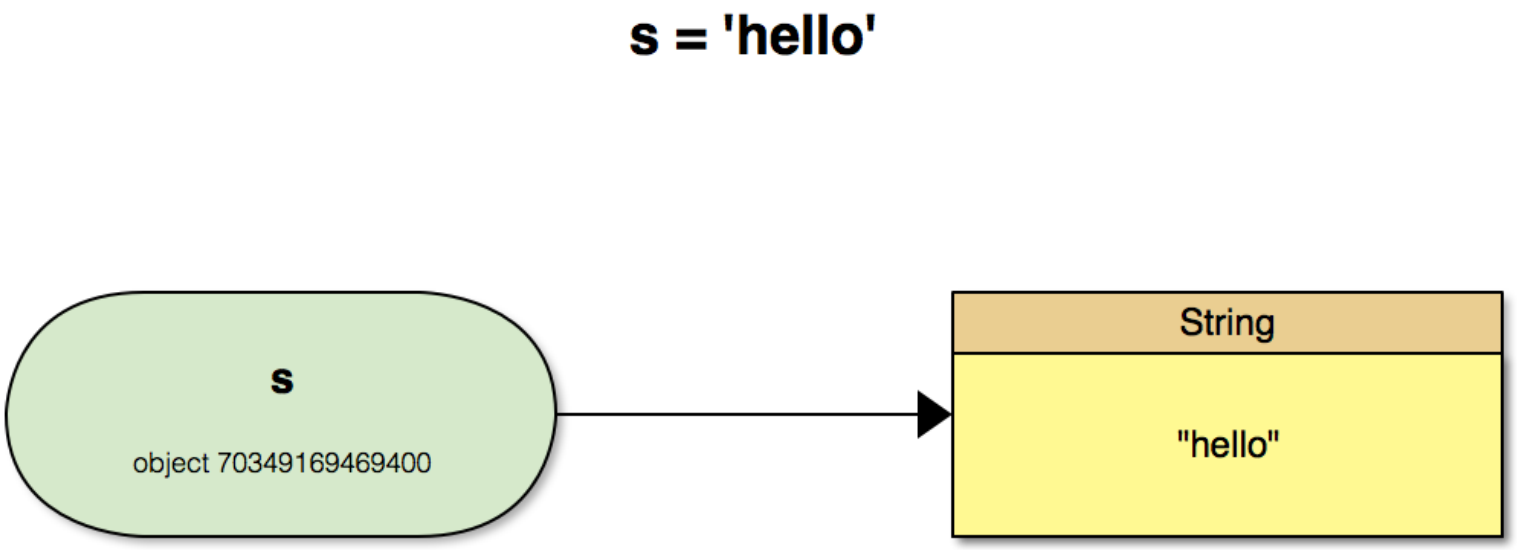
If you run this code, you will see something like this:

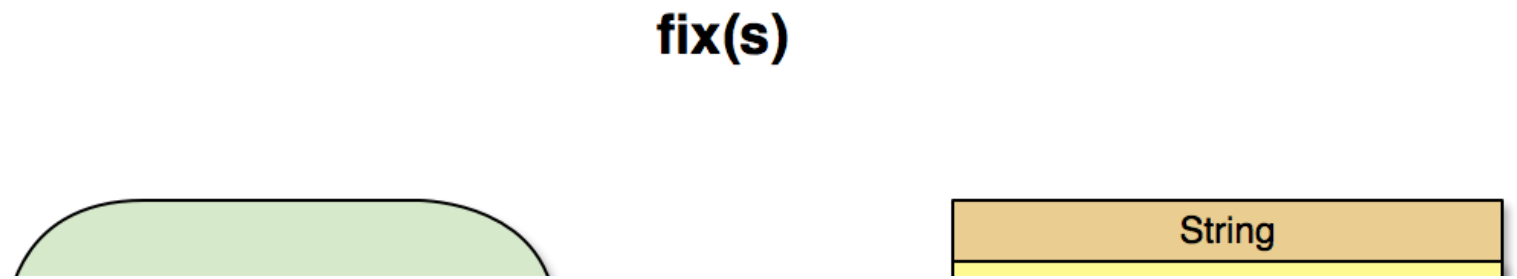
```
1 original object 70349169469400
```

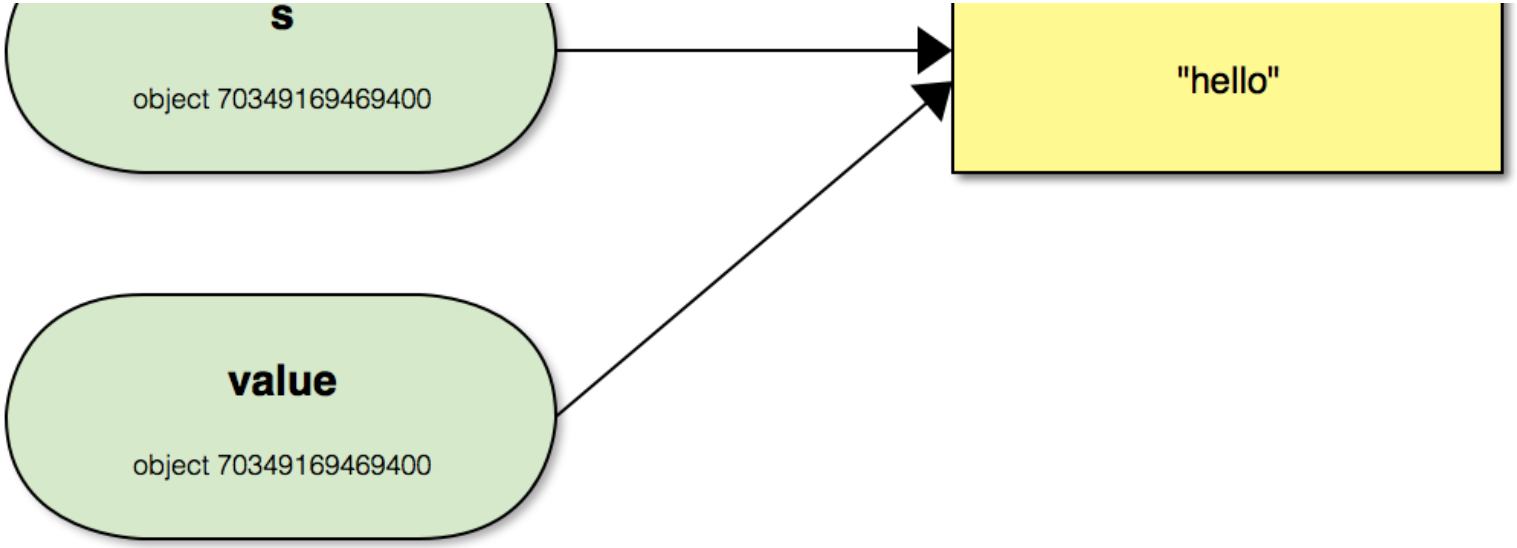
```
2 initial object 70349169469400
3 upcased object 70349169435840
4 final object 70349169435840
```

This shows that `value = value.upcase` bound the return value of `value.upcase` to `value`; `value` now references a different object than it did before. Prior to the assignment, `value` referenced the same String as referenced by `s`, but after the assignment, `value` references a completely new String; the String referenced by `#upcase`'s return value.

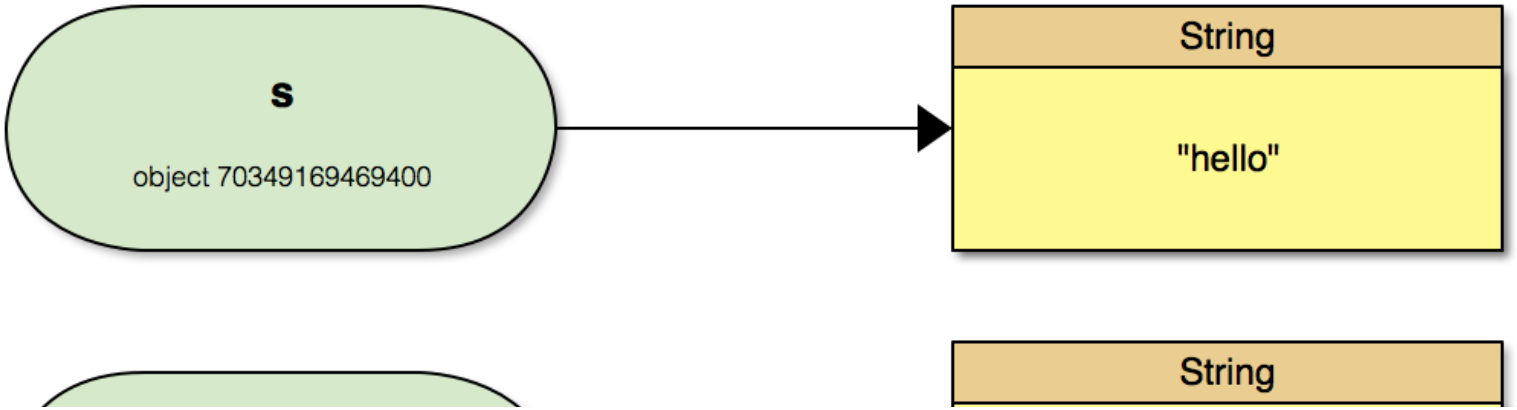
Pictorially:

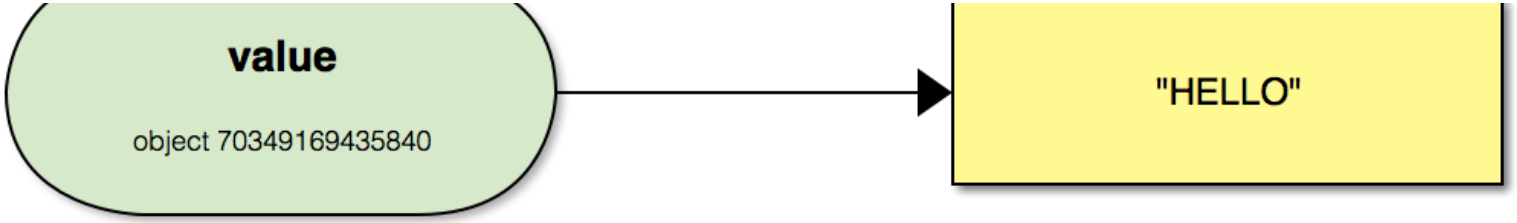




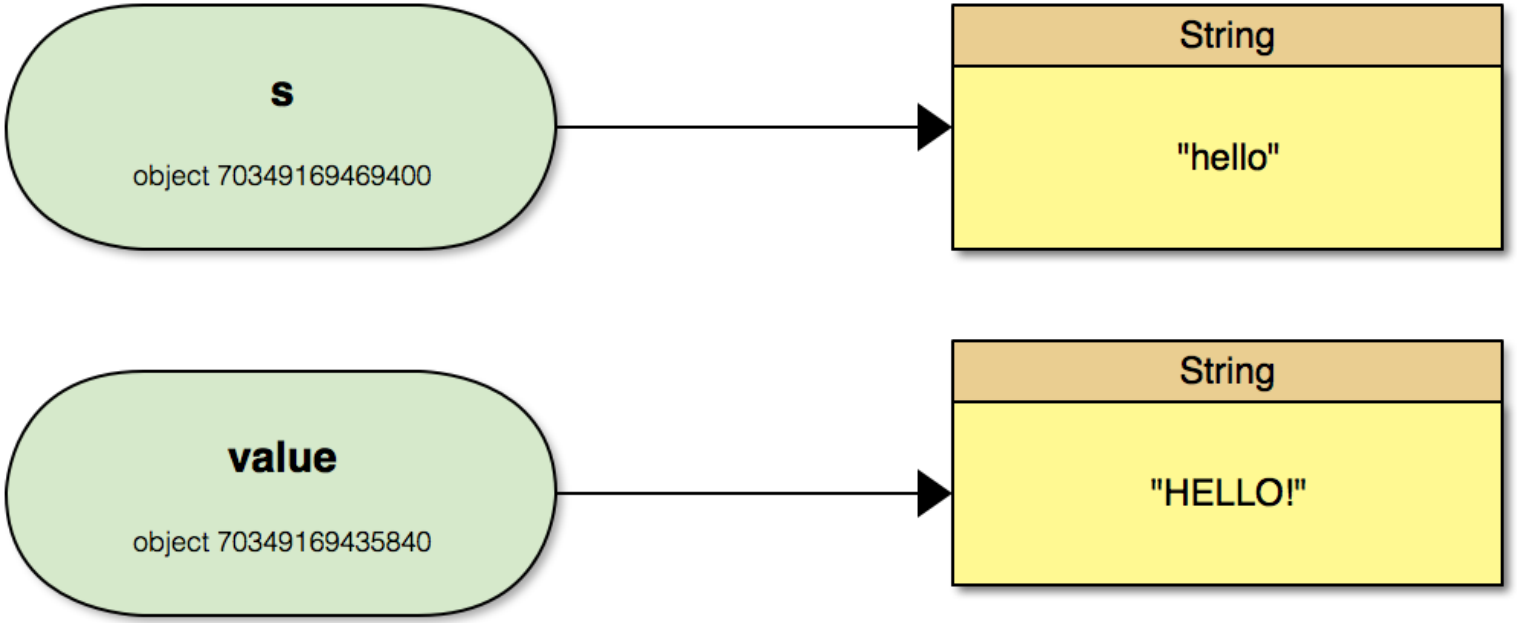


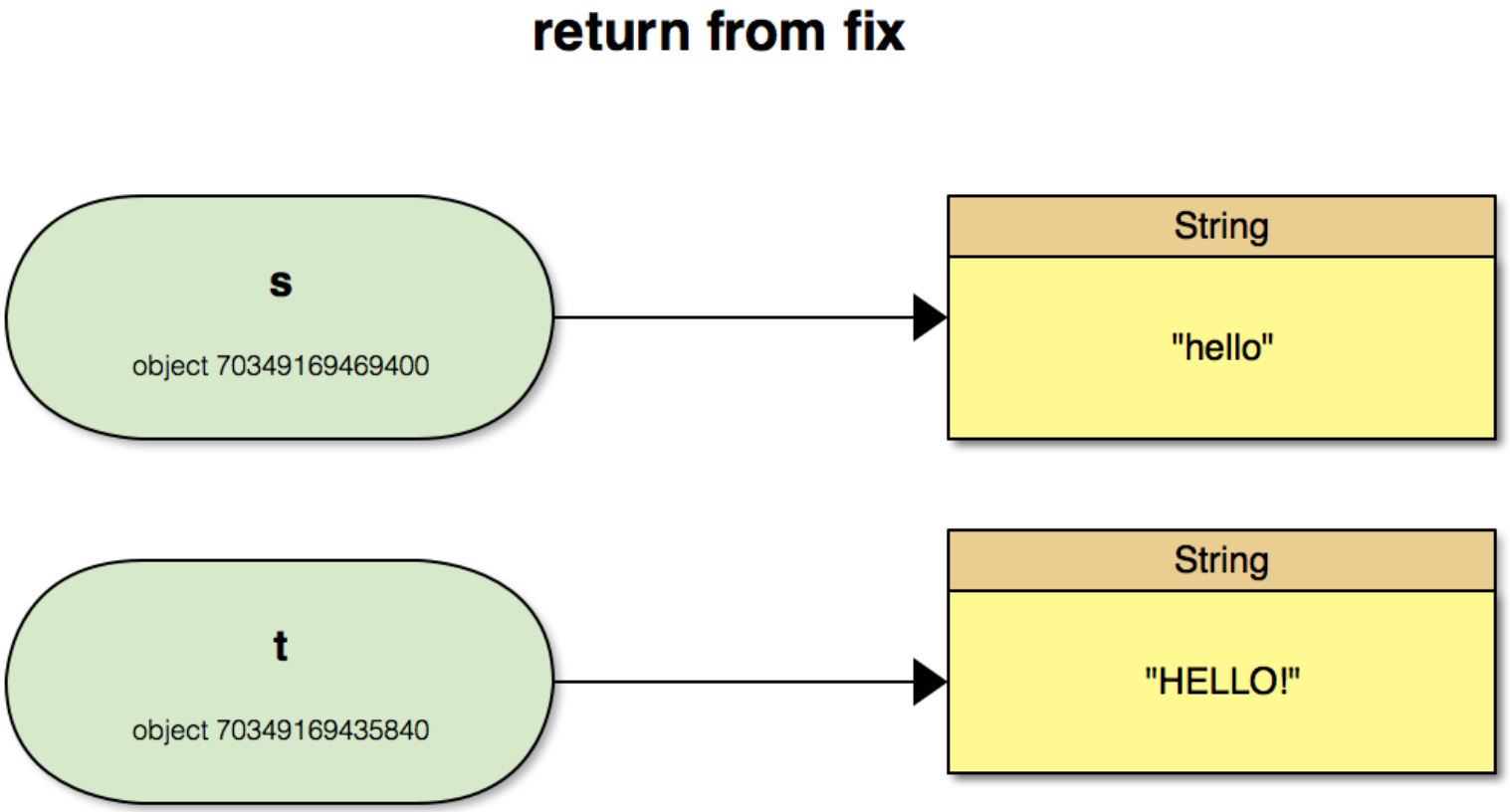
value = value.upcase





value.concat('!')





Assignment always binds the target variable on the left hand side of the = to the object referenced by the right hand side. The object originally referenced by the target variable is never modified.

However, be aware that any mutating operations prior to the assignment may still take place:

```
1 def fix(value)
```

```
2   value << 'xyz'
3   value = value.upcase
4   value.concat('!')
5 end
6 s = 'hello'
7 t = fix(s)
```

This program modifies the original string so its value is `helloxyz`. However, thanks to the assignment, it is not changed to `HELLOXYZ` or `HELLOXYZ!`; those changes occur in a different object that gets returned by the method.

These types of issues arise not only with assignment, but also with assignment operators like `*=`, `+=`, and `%=`. These are all implemented in terms of assignment, and that assignment always causes the target to reference a possibly different object. None of these operations mutate their operands.

This can be confusing at times. For instance:

```
1  >> s = 'Hello'
2  => "Hello"
3
4  >> s.object_id
5  => 70101471465440
6
7  >> s += ' World'
8  => "Hello World"
9
10 >> s
11 => "Hello World"
12
13 >> s.object_id
14 => 70101474966820
```

Though it looks as if we are modifying `s` when we write `s += ' World'`, we are actually creating a brand-new String with a new object id, and then binding `s` to that new object. We can see by looking at the object ids that a new object is created.

If you are new to Ruby, this **will** trip you up. It's guaranteed. It's probably already happened; it's likely why you are reading this article.

> Setter methods for class instance variables and indexed assignment are **not** the same as assignment. We'll return to this below, but for now, remember that setter methods and indexed assignment usually mutate the calling object.

Note the word "possibly" in "causes the target to reference a possibly different object". The reason for this can be seen by running yet another variation on our `#fix` method:

```
1  >> def fix(value)
2  --   value = value.upcase!
3  --   value.concat('!')
4  -- end
5  => :fix
6
7  >> s = 'hello'
8  => "hello"
9
10 >> s.object_id
11 => 70363946430440
12
13 >> t = fix(s)
14 => "HELLO!"
15
```

```
16 >> s
17 => "HELLO!"
18
19 >> t
20 => "HELLO!"
21
22 >> s.object_id
23 => 70363946430440
24
25 >> t.object_id
26 => 70363946430440
```

This time, though we assigned a reference to `value`, we end up with both `s` and `t` referring to the same object. The reason for this is that `String#upcase!` returns a reference to its caller, `value`. Since the reference returned by `value.upcase!` is the same, albeit modified, String we started with, the assignment effectively rebinds `value` back to the object it was previously bound to; nothing is changed by the assignment.

Mutating Methods

A method is said to be mutating with respect to an argument or its caller if it modifies it.

Consider the `String#strip!` method that removes leading and trailing whitespace from a String object:

```
1  >> s = '   hey   '
2  => "   hey   "
3
4  >> s.object_id
5  => 70101479494960
6
7  >> s.strip!
8  => "hey"
9
10 >> s.object_id
11 => 70101479494960
```

Here, we mutate the original String object; `s` references the same object both before and after `#strip` is called. Only the state of the object has been changed.

Many, but not all, methods that mutate their caller use `!` as the last character of their name. However, this is not guaranteed to be the case. For instance, `String#concat` is a mutating method, but it does not include a `!`.

There are several common methods that sometimes cause confusion, `#[]=`, `#<<`, and setter methods.

Indexed Assignment is Mutating

Indexed assignment, such as that used by String, Hash, and Array objects can be confusing:

```
1 str[3] = 'x'
2 array[5] = Person.new
3 hash[:age] = 25
```

This looks exactly like assignment, which is non-mutating, but is, in fact, mutating. `#[]` modifies the original object (the String, Array, or Hash). It doesn't change the binding of each variable.

Consider this code:

```
1  def fix(value)
2    value[1] = 'x'
3    value
4  end
5
6  s = 'abc'
7  t = fix(s)
8  p s            # "axc"
9  p t            # "axc"
10 p s.object_id  # 70349153406320
11 p t.object_id  # 70349153406320
```

Earlier, we saw similar code that merely assigned to `value`, and we saw that performing assignment bound `value` to a completely new String. Thus, `s` and `t` referenced different objects.

Here, though, we are using indexed assignment instead, and, perhaps surprisingly, the binding does not change. Even after the assignment to `value[1]`, `value` still references the same (albeit mutated) String object.

The reason for this is that indexed assignment is a method that a class must supply if it needs indexed assignment. This method is named `#[]=`, and `#[]=` is expected to mutate the object to which it applies. It does not create a new object.

Let's examine this with an Array:

```
1  >> a = [3, 5, 8]
2  => [3, 5, 8]
3
4  >> a.object_id
5  => 70240541515340
6
7  >> a[1].object_id
8  => 11
9
10 >> a[1] = 9
11 => 9
12
13 >> a[1].object_id
14 => 19
15
16 >> a
17 => [3, 9, 8]
18
19 >> a.object_id
20 => 70240541515340
```

Here, we can see that we have mutated the Array `a` by assigning a new value to `a[1]`, but have not created a new Array. `a[1] = 9` isn't assigning anything to `a`; it is assigning `9` to `a[1]`; that is, this assignment changes `a[1]` so that it references the new object `9`. You can see this by looking at `a[1].object_id` both before and after the assignment. Despite this change, though, `a` itself still points to the same (now mutated) Array we started with.

This is normal behavior when working with objects that support indexed assignment: the assignment does cause a new reference to be made, but it is the collection element e.g., (`a[1]`) that is bound to the new object, not the collection (enclosing object) itself.

Concatenation is Mutating

The `#<<` method used by collections like Arrays and Hashes, as well as the String class, implements concatenation; this is very similar to the `+=` operator. However, there is a major difference; `+=` is non-mutating, but `#<<` is mutating. Lets look at an example that uses `String#<<`:

```
1  >> s = 'Hello'
```

```
2  => "Hello"
3
4  >> s.object_id
5  => 70101471465440
6
7  >> s << ' World'
8  => "Hello World"
9
10 >> s
11 => "Hello World"
12
13 >> s.object_id
14 => 70101471465440
```

This example is nearly identical to our earlier example using +=, but with one major difference: we use #<< instead of +=. The #<< method is mutating with respect to its caller (`s` here), so the object referenced by `s` is modified; no new objects are created, so `s` still references the same object it did prior to the #<< call.

The << operator is actually a method that is defined for some classes. It is usually used as a shorthand for appending new values to a collection or String. Such classes define << to mutate their left-hand operand (the caller).

> Some classes use << for "bit shift" operations; such operations are usually non-mutating. Other classes may employ << for operations that have nothing to do with bit shifts or appending; in those cases, you likely need to read the documentation or test the operation in a short program to determine if it is mutating or non-mutating.

Setters are Mutating

Setters are very similar to indexed assignment; they are methods that are defined to modify the state of an object. Both employ the `something = value` syntax, so they superficially look like assignments. With indexed assignment, the elements of a collection (or the characters of a String) are replaced; with setters, the state of the object is modified, usually by modifying an instance variable.

Setter invocation looks like this:

```
1 person.name = 'Bill'
2 person.age = 23
```

This looks exactly like assignment, which is non-mutating, but, since these are setter calls, they actually mutate the object bound to `person`.

We won't go into a lot of detail to illustrate this; suffice to say that a detailed discussion would be nearly identical to the discussion of indexed assignment.

It's possible to define setter methods that don't mutate the original object. Such setters should still be treated as mutating since they don't create new copies of the original object.

Refining the Mental Model

What does this have to do with whether Ruby is pass by value or pass by reference? The mere fact that Ruby can have methods that mutate its arguments would seem to say that Ruby must use pass by reference in some circumstances. Arguments that are passed by copy cannot be mutated, so Ruby must use pass by reference when a method can mutate its arguments.

More importantly, the question of whether Ruby is pass by value or pass by reference usually concerns whether a method will mutate its arguments or caller. With this discussion, we're better equipped to determine whether a method will mutate one of them.

The presence of a `!` at the end of a method name is a pretty good indicator that a method mutates its caller. However, not all mutating methods use the `!` convention. In such cases, you need to look at the source code of the method to see what operations are performed. Certain operations, like setters and indexed assignments should always be treated as mutating methods; others, like assignment and the assignment operators (+=, *=, etc) are always non-mutating.

While none of this modifies our mental model for object passing, it is all consistent with that mental model. Immutable objects still seem to be passed by value, while mutable objects seemed to be passed by reference. What we have done, though, is show that assignment can break the binding between an argument name and the object it references. This is important to keep in mind when examining the relationships between variables and objects.

Conclusion

In this article, we've seen that methods in Ruby can be mutating or non-mutating with respect to individual arguments, to include the caller. A method that does not modify its arguments or caller is non-mutating with respect to those objects; a method that does modify some of them is mutating with respect to those modified objects.

We've also learned that assignment in Ruby acts like a non-mutating method — it doesn't modify any objects, but does alter the binding for the target variable. However, the syntactically similar indexed assignment and object setter operations are mutating. We've also seen that the #<< operator — when used for concatenation operations — is mutating, while the very similar operation performed by += is non-mutating.

We're now ready to dive more deeply into the topic of whether Ruby uses pass by value or pass by reference. Continue reading at Object Passing in Ruby – by Reference or by Value? article.

Jul 24th, 2016 • by Pete Hanson

Get Our Newsletter

Beginner focused news, tutorials and coding tips. We won't spam or waste your time.

You have subscribed successfully Please enter a valid email address There was a problem, please try again

Your email address Subscribe

See What Our Community is Talking About

From soft topics to in-depth tutorials, members of the Launch School community have all sorts of useful information to share.

Community Publications on Medium

Share on

Popular Posts

- Our Secret Sauce: Mastery Based Learning
- Meet a Student Podcast Series
- A Glimpse Into the Launch School Curriculum
- How to Launch a Career in Software Development
- The "Beginning Ruby" Live Session Series (6 parts)

- About Us
- Faq