Variable References and Mutability of Ruby Objects

This is the first 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. You can find many such discussions in articles that attempt to answer the question "Is Ruby pass by reference or pass by value?" Our goal in this series isn't necessarily to answer this question — though we will provide an answer, of sorts — but to put that question in the context of how Ruby actually works.

The answer to the question really isn't that important. What is important is more fundamental: knowing how Ruby works. A proper understanding of how Ruby works will take you much further than being able to provide an answer to the question; it will help you learn to anticipate what Ruby will do, when it will do it, and why it will do it. This will help you become a better Ruby programmer, and reduce the number of bugs you encounter due to unexpected behaviors.

In this article, we start out by exploring how Ruby variables and objects are connected to each other, and the part that mutability or immutability of objects plays when manipulating variables. We will also introduce the concepts of pass by reference and pass by value, but a more detailed discussion will be postponed until the article Object Passing in Ruby – by Reference or by Value.

Introduction

It seems simple: you take an object in Ruby, and you assign it to a variable. Subsequently, that variable can be used to access and manipulate the object. It's the same in dozens of different programming languages. Can Ruby be any different? What kind of complexity can lurk behind such a simple concept?

As simple as the idea of variables seems, there are differences between languages. Some languages, like C++ and Perl, make copies of an object when you assign them to a variable; others, like Javascript and Python, create a link — a reference or binding — between the variable and the object in question. These differences in behavior can lead to some unexpected behavior if you don't understand how your favorite language does things.

Variables and References

An object is a bit of data that has some sort of state — sometimes called a value — and associated behavior. It can be simple, like the Boolean object `true`, or it can be complex, like an object that represents a database connection.

Objects can be assigned to variables, like this:

```
1 >> greeting = 'Hello'
2 => "Hello"
```

This tells Ruby to associate the name `greeting` with the String object whose value is "Hello". In Ruby, `greeting` is said to *reference* the String object. We can also talk of the variable as being *bound* to the String object, or binding variable to the String object. Internally, the relationship looks like this:

Here, the String object represented by the literal “Hello” is assigned to a variable that has the name `greeting`. This causes the variable `greeting` to reference the String object whose value is “Hello”. It does so by storing the object id of the String. Subsequently, we can simply use the value “Hello” by using the variable `greeting`:

```
1 >> greeting
2 => "Hello"
3
4 >> greeting.object_id
5 => 70101471431160
```

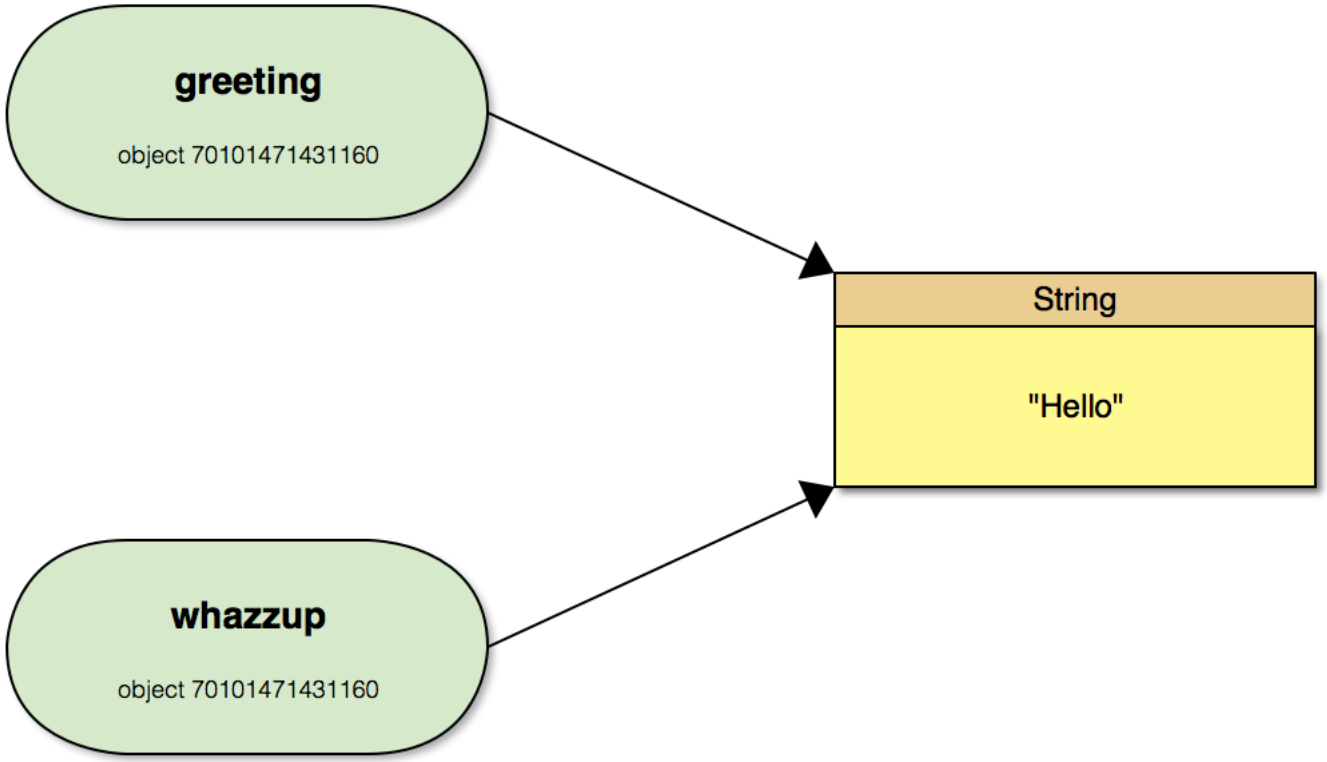
> We use `#object_id` frequently in this article. Every object in Ruby has a unique object id, and that object id can be retrieved simply by calling `#object_id` on the object in question. Even literals, such as numbers, booleans, `nil`, and Strings have object ids:
>
> ```
> 1  >> 5.object_id
> 2  => 11
> 3
> 4  >> true.object_id
> 5  => 20
> 6
> 7  >> n
> 8  nil.object_id
> 9  => 8
> 10
> 11 >> "abc".object_id
> 12 => 70101471581080
> ```
>
> Get comfortable with using `#object_id`, both while reading this article, and whenever you have trouble understanding why an object has an unexpected value.

Let’s assign `greeting` to a new variable:

```
1  >> whazzup = greeting
2  => "Hello"
3
4  >> greeting
5  => "Hello"
6
```

```
7  >> whazzup
8  => "Hello"
9
10 >> greeting.object_id
11 => 70101471431160
12
13 >> whazzup.object_id
14 => 70101471431160
```

Internally, the situation now looks like this:

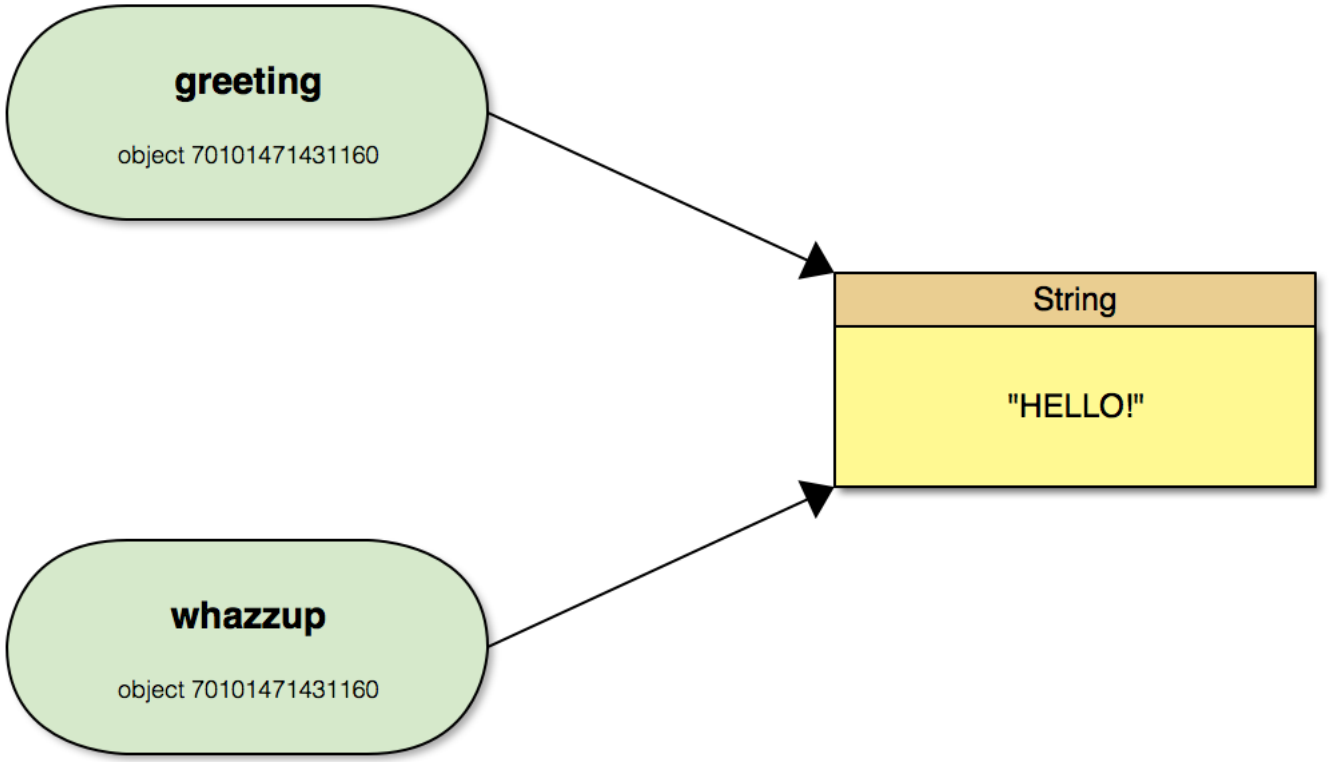


While the exact value returned by `#object_id` on your computer may differ from what we show, you should see that both object ids are the same. This demonstrates that both `greeting` and `whazzup` not only reference a String with the same value, but are, in fact, references to the same String; `greeting` and `whazzup` are aliases for each other. We can show this by using one of the two variables to change the object:

```
1  >> greeting.upcase!
2  => "HELLO"
```

```
3
4  >> greeting
5  => "HELLO"
6
7  >> whazzup
8  => "HELLO"
9
10 >> whazzup.concat('!')
11 => "HELLO!"
12
13 >> greeting
14 => "HELLO!"
15
16 >> whazzup
17 => "HELLO!"
18
19 >> greeting.object_id
20 => 70101471431160
21
22 >> whazzup.object_id
23 => 70101471431160
```

Since both variables are associated with the same object, using either variable to alter the object is reflected in the other variable. We can also see that the object id does not change. Internally, we now have this:



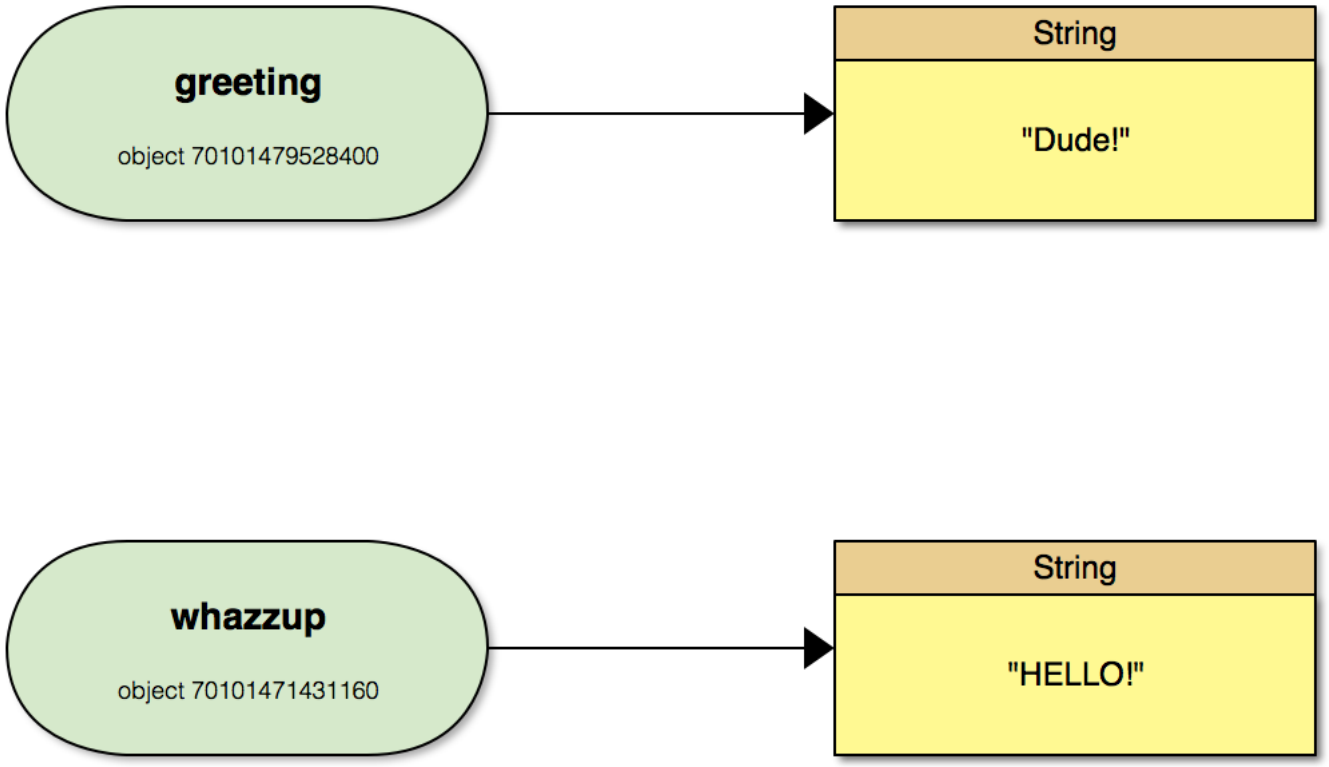
Reassignment

Let's assign a new object to one of these variables:

```
1  >> greeting = 'Dude!'
2  => "Dude!"
3
4  >> puts greeting
5  => "Dude!"
6
7  >> puts whazzup
8  => "HELLO!"
9
10 >> greeting.object_id
11 => 70101479528400
12
```

```
13 >> whazzup.object_id
14 => 70101471431160
```

Here, we see that `greeting` and `whazzup` no longer refer to the same object; they have different values and different object ids. Crazy, right? Internally, we now have:



What this shows is that reassignment to a variable doesn't change the object referenced by that variable; instead, the variable is bound to a completely new object — made to reference a new object. The original object is merely disconnected from the variable. In this example, `greeting` is bound to the String object whose value is `Dude!`, while `whazzup` continues to reference the String object whose value is `HELLO!`.

Mutability

Now that we understand what references are and how they relate to variables, we can investigate *mutability*. Objects can be either mutable or immutable. Mutable objects can be changed; immutable objects cannot be changed.

Some objects can be modified, others can never be modified; different languages, again, do different things. In C++ and Perl, for instance, string objects are mutable, but in Java and Python, string objects are immutable. Understanding mutability of an object is necessary to understanding how your language deals with those objects.

Immutable Objects

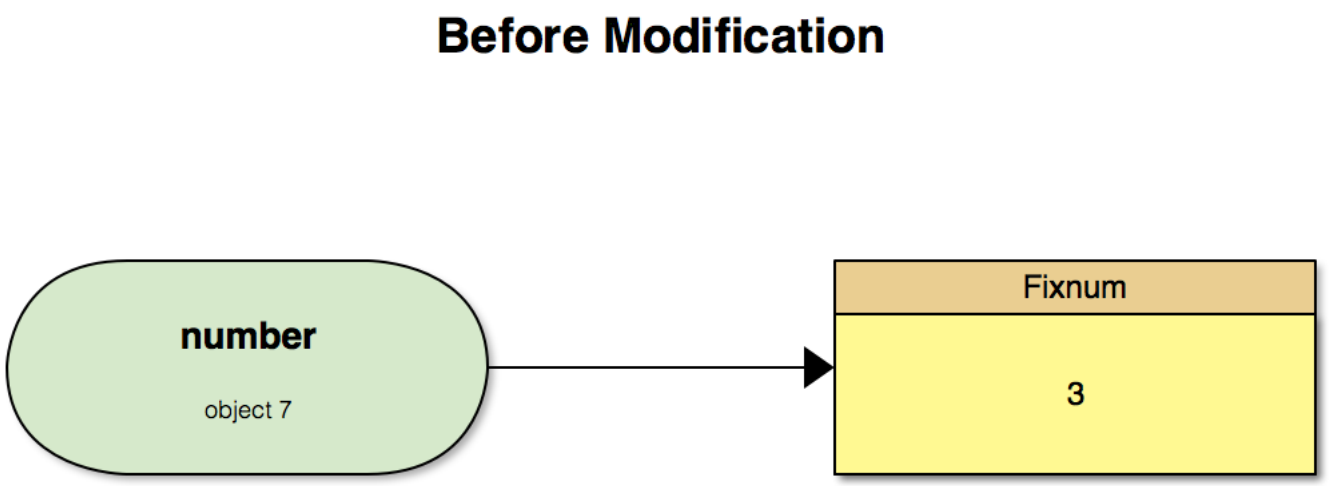
In Ruby, numbers and boolean values are immutable. Once we create an immutable object, we cannot change it. “But,” we hear you ask, “What about this code?”

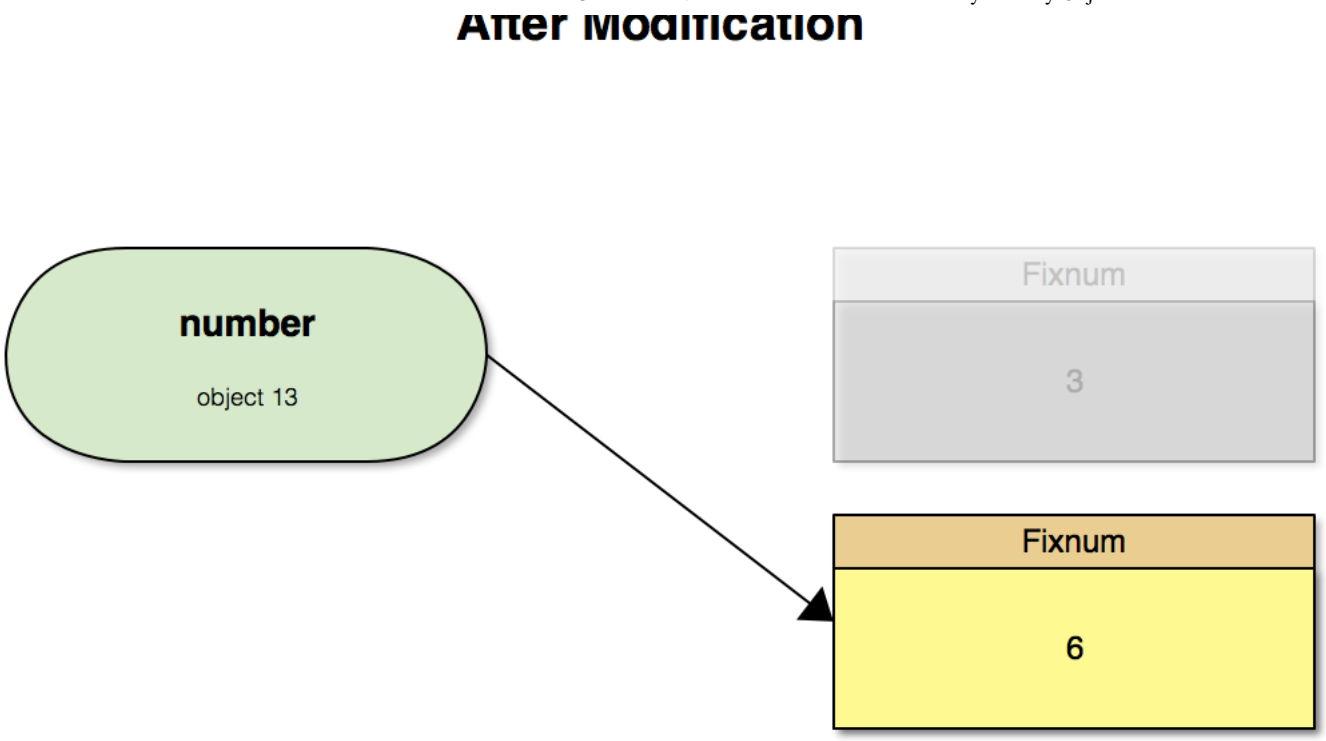
```
1  >> number = 3
2  => 3
3
4  >> number
5  => 3
6
7  >> number = 2 * number
8  => 6
9
10 >> number
11 => 6
```

“Doesn’t this show that the object `3` was changed to `6`?”

Nope. As we saw above, this is reassignment which, as we learned, doesn’t change the object. Instead, it binds a new object to the variable. In this case, we create a new Integer with a value of `6` and assign it to `number`. There are, in fact, no methods available that let you alter the value of any immutable object. All you can do is reassign the variable so it references a different object. This disconnects the original object from the variable, which makes it available for garbage collection unless another reference to the object exists elsewhere.

Internally, the reasssignment looks like this:





Lets demonstrate this in irb:

```
1  >> a = 5.2
2  => 5.2
3
4  >> b = 7.3
5  => 7.3
6
7  >> a
8  => 5.2
9
10 >> b
11 => 7.3
12
13 >> a.object_id
14 => 46837436124653162
15
16 >> b.object_id
17 => 65752554559609242
```

We start by assigning the Float values `5.2` and `7.3` to the variables `a` and `b`. We see these values and their object ids when we ask irb to display them. The object ids differ, so the variables reference different objects.

Lets see what happens if `b` is assigned to `a`:

```
1  >> a = b
2  => 7.3
3
4  >> a
5  => 7.3
6
7  >> b
8  => 7.3
9
10 >> a.object_id
11 => 65752554559609242
12
13 >> b.object_id
14 => 65752554559609242
```

irb now displays the same value for each variable. More interestingly, it shows that the object ids for both `a` and `b` are the same. The object that originally held the value `5.2` is now gone.

Let's try to change the object now:

```
1  >> b += 1.1
2  => 8.4
3
4  >> a
5  => 7.3
6
7  >> b
8  => 8.4
9
10 >> a.object_id
11 => 65752554559609242
12
13 >> b.object_id
14 => 32425917317067566
```

On the first line, we try to change the object referenced by `b` by incrementing `b` by `1.1`. This yields 8.4 and, as we can see, `b` is also `8.4`. `a` has not been modified, and still references the `7.3` object. But, `b` now references a completely new object. Though we changed the value associated with `b`, we didn't modify the object — the object is immutable. Instead, `+=` created a brand-new Float object, and bound `b` to the new object.

Even simple assignment doesn't change an immutable object:

```
1 >> a = 5.5
2 => 5.5
3
4 >> a.object_id
5 => 49539595901075458
```

Instead of modifying the original object, a new Float is created and `a` is bound to the new object.

Immutable objects aren't limited to numbers and booleans. Objects of some complex classes, such as `nil` (the only member of the NilClass class) and Range objects (e.g., `1..10`) are immutable. Any class can establish itself as immutable by simply not providing any methods that alter its state.

Mutable Objects

Unlike numbers, booleans, and a few other types, most objects in Ruby are mutable; they are objects of a class that permit modification of the object's state in some way. Whether modification is permitted by setter methods or by calling methods that perform more complex operations is unimportant; as long as you can modify an object, it is mutable.

Consider Ruby Array objects; you can easily modify elements using indexed assignment:

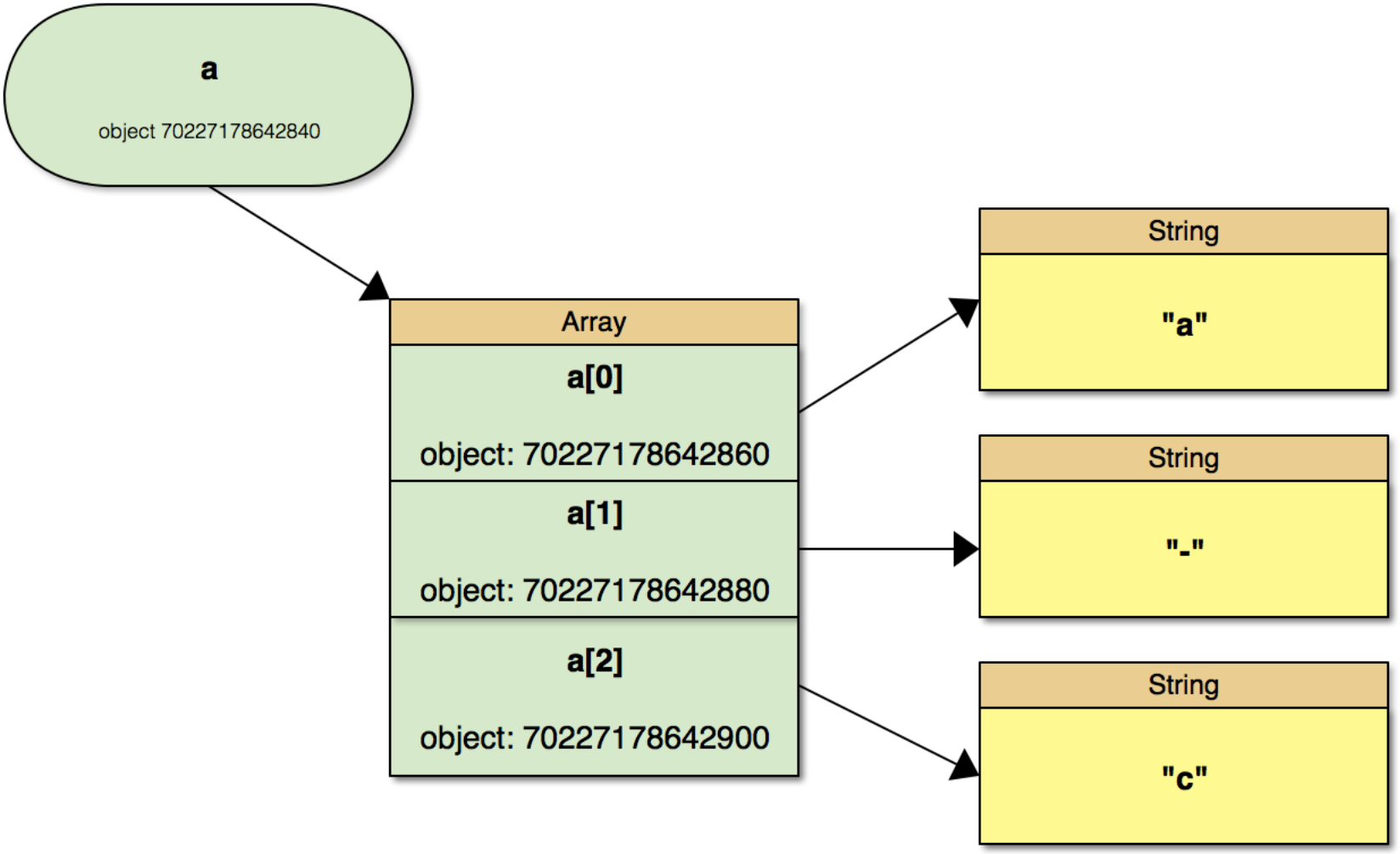
```
1  >> a = %w(a b c)
2  => ["a", "b", "c"]
3
4  >> a.object_id
5  => 70227178642840
6
```

```
>> a[1] = '-'
=> "-"

>> a
=> ["a", "-", "c"]

>> a.object_id
=> 70227178642840
```

This demonstrates that we can modify the value of `a`, but it doesn't create a new object since the object id remains the same. We can see why this is by looking at how `a` is stored in memory:


a
object 70227178642840
Array
a[0]
object: 70227178642860
a[1]
object: 70227178642880
a[2]
object: 70227178642900
String
"a"
String
"-"
String
"c"

Here, we see that `a` is a reference to an Array, and, in this case, that Array contains three elements; each element is a reference to a String object. When we assign `-` to `a[1]`, we are binding `a[1]` to a new String; we're modifying `a`, but not `a[1]`.

Strings and other collection classes are similar in the way they behave — variables reference the collection (or String), and the collection contains references to the actual objects in the collection. Strings are a little bit different — it's not really necessary to have separate objects for each character — but they act in a similar way.

Several Array methods, such as `#delete`, `#fill`, and `#insert` mutate the original object without creating a new one.

A Brief Introduction to Object Passing

When you pass an object as an argument to a method, the method can — in theory — either modify the object, or leave it unmodified. It's easy enough to see that any method can avoid modifying its arguments. However, whether or not the method can modify an argument is less clear; the ability to modify arguments depends in part on the mutability or immutability of the object represented by the argument, but also on how the argument is passed to the method.

Some languages make copies of method arguments, and pass those copies to the method — since they are merely copies, the original objects can't be modified. Objects passed to methods in this way are said to be *passed by value*, and the language is said to be using a *pass by value* object passing strategy.

Other languages pass references to the method instead — a reference can be used to modify the original object, provided that object is mutable. Objects passed to methods in this way are said to be *passed by reference*, and the language is said to be using a *pass by reference* object passing strategy.

Many languages employ both object passing strategies. One strategy is used by default; the other is used when a special syntax, keyword, or declaration is used. Some languages may even employ different defaults depending on the object type — for example, numbers may be passed using a pass by value strategy, while strings may be passed using a pass by reference strategy.

Regardless of which strategy a language employs for a given argument and method, it's important to know which one is used so you can understand what happens if the method modifies one of its arguments.

Developing A Mental Model

When learning new concepts, it often helps to develop a mental model of the concept, and then refine that model as additional information comes to light. We're now in a position to begin formulating our mental model for object passing in Ruby.

Pass by value, as you'll recall, means copying the original objects, so the original object cannot be modified. Since immutable objects cannot be changed, they act like Ruby passes them around by value. This isn't a completely accurate interpretation of how Ruby passes immutable objects, but it helps us determine why the following code works like it does:

```
1 def increment(a)
2   a = a + 1
3 end
4
5 b = 3
6 puts increment(b)    # prints 4
7 puts b               # prints 3
```

Here, the numeric object `3` is immutable. You can reasonably say that `b` is not modified by `#increment` since `b` is passed by value to `#increment` where it is bound to variable `a`. Even though `a` is set to `4` inside the method, and returned to the caller, the original object referenced by `b` is untouched.

Mutable objects, on the other hand, can always be modified simply by calling one of their mutating methods. They act like Ruby passes them around by reference; it isn't necessary for a method to modify an object that is passed by reference, only that it can modify the object. As you'll recall, pass by reference means that only a reference to an object is passed around; the variables used inside a method are bound to the original objects. This means that the method is free to modify those objects. Once again, this isn't completely accurate, but it is helpful. For instance:

```
1 def append(s)
2   s << '*'
3 end
4
5 t = 'abc'
6 puts append(t)    # prints abc*
7 puts t            # prints abc*
```

Here, the String object `abc` is mutable. You can reasonably say that `t` is modified by `#append` since `t` is passed by reference to `#append` where it is bound to variable `s`. When `s` is modified by `append`, it modifies the same object referenced by `t`, so upon return from the method, `t` still refers to the original (albeit modified) String object.

Conclusion

In this article, we've seen that Ruby variables are merely references to objects in memory; that is, a variable is merely a name for some object. Multiple variables can reference the same object, so modifying an object using a given variable name will be reflected in every other variable that is bound to that object. We've also learned that assignment to a variable merely changes the binding; the object the variable originally referenced is not modified. Instead, a different object is bound to the variable.

We've also learned that certain object types, primarily numbers and Booleans but also some other types, are immutable in Ruby — unchanging; many other objects are mutable — changeable. If you attempt to change an immutable object, you won't succeed — at best, you can create a new object, and bind a variable to that object with assignment. Mutable objects, however, can be modified without creating new objects.

Finally, we've learned a bit about what pass by value and pass by reference mean. We've established a mental model that says that Ruby is pass by value for immutable objects, pass by reference otherwise. This model isn't perfect, but it can be used to help determine whether the object associated bound to an argument will be modified.

We're now equipped with the tools we need to explore the differences between mutating and non-mutating methods. Continue reading at Ruby's Mutating and Non-Mutating Methods.

Jul 24th, 2016 • by Pete Hanson