
## Blog

- RSS

SEARCH

# Object Passing in Ruby - Pass by Reference or Pass by Value

This is the last 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. If you haven't read the first two articles, you may want to check them out first: Understand Variable References and Mutability and Ruby's Mutating and Non-Mutating Methods.

We now have a good grip on how ruby uses variables to reference objects, what the terms mutability and immutability mean, and what it means for a method to be mutating or non-mutating. We've also been briefly introduced to the concept of object passing, and have established an initial mental model that states that ruby appears to use pass by value for immutable objects, and pass by reference for mutable objects. We've also established that assignment does not mutate objects but instead binds variables to new objects, while setter methods and indexed assignment do mutate objects.

You might have noticed that we've been careful to say "appears to use" instead of "uses"; there's a good reason for this, which we'll illuminate below.

## What is Object Passing?

New developers usually run into the terms *pass by reference* and *pass by value* sooner instead of later. These topics come up often when learning a new language, and when trying to understand how data is passed around by the language. Specifically, a developer needs to know what happens to the original objects passed to or returned from a method. In this article, we will explore what these terms mean, and how they apply to our favorite language, ruby.

In ruby, nearly everything is an object. When you call a method with some expression as an argument, that expression is evaluated by ruby and reduced, ultimately, to an object. The expression can be an object literal, an variable name, or a complex expression; regardless, it is reduced to an object. Ruby then makes that object available inside the method. This is called passing the object to the method, or, more simply, *object passing*.

In addition to method arguments, the *caller* (sometimes called the receiver) of a method call — the object on which the method is called — can be thought of as an implied argument. As such, we need to include method callers in our discussion of object passing.

We also need to think about return values. Just as much as arguments are passed to methods, return values are passed by those methods back to the caller. Thus, return values must be included in our discussion of object passing.

Ruby also supports blocks, procs, and lambdas. All of these include the concepts of passing arguments and return values around. We will usually talk of passing objects to and from methods, but you should interpret that as referring to blocks, procs, and lambdas as well.

In ruby, many operators like `+`, `*`, `[]`, and `!` are methods, and even `=` acts like a method. This means that the operands of these operators are arguments, and the operators have return values; these arguments and return values are passed around just like other methods.

Because of all of this generality, we will use some terminology pretty loosely. Objects can be literals, named objects (variables and constants), or complex expressions. Methods can include methods, blocks, procs, lambdas, and even operators. Arguments can include actual arguments, the caller of the method, operator operands, or a return value. This loose use of the terminology is imprecise, but easier to understand than repeating ourselves at every opportunity.

# Evaluation Strategies

Every computer programming language uses some sort of *evaluation strategy* when passing objects. This strategy determines when expressions are evaluated, and what a method can do with the resulting objects. The most common strategies are known as *strict evaluation* strategies. With strict evaluation, every expression is evaluated and converted to an object before it is passed along to a method. Ruby uses strict evaluation exclusively.

The two most common strict evaluation strategies are pass by value and pass by reference. Collectively, we will refer to pass by value and pass by reference as *object passing strategies*.

## Why is the Object Passing Strategy Important?

Most computer languages that employ strict evaluation use pass by value by default. Most of those languages also make it possible to pass by reference when needed. Few languages are purely pass by value or pass by reference. Understanding which strategy is used (and when) is key to understanding what happens to an object that gets passed to a method. For example, if the method does something that appears to change the object, is that change local to the method, or does it result in changes to the original object? This is crucial in code like this:

```
1 def increment(x)
2   x << 'b'
3 end
4
5 y = 'a'
6 increment(y)
7 puts y
```

Just by reading, can you tell whether this code will output `a` or `ab`? Hypothetically, if ruby is pass by value, this code prints `a`. The reason for this is that a pass by value strategy creates a copy of `y` before passing it to `#increment`; since `#increment` has only a copy of `y`, it can't actually modify `y`.

However, if ruby is pass by reference, this code prints `ab`. Here, ruby passes a reference to `y` to `#increment`, so `x` becomes an alias for `y`. When you modify `x`, you also modify the aliased object, `y`.

The fact that the object passing strategy determines what this code prints should show you why knowledge of the strategy is important. It might be easy to look at this code and decide whether ruby uses pass by value or pass by reference. If you run it, it will print `ab`, implying that ruby is pass by reference. However, as we'll see, this answer is too simplistic; there's a lot more going on than this simple example demonstrates.

## Pass by Value

With pass by value, a copy of an object is created, and it is that copy that gets passed around. Since it is merely a copy, it is impossible to change the original object; any attempt to change the copy just changes the copy and leaves the original object unchanged.

Passing around immutable values in ruby acts a lot like pass by value:

```
1 def plus(x, y)
2   x = x + y
3 end
4
5 a = 3
6 b = plus(a, 2)
7 puts a # 3
8 puts b # 5
```

As you can see, although we assign a new value to `x` in `#plus`, the original argument, `a`, is left unchanged. (The method, however, does return the result of adding `2` to `a`, `5`, which is stored in `b`.) So, you can say that ruby appears to be pass by value, at least with respect to immutable values.

### Pass by Reference

By contrast, with pass by reference, a *reference* to an object is passed around. This establishes an alias between the argument and the original object: both the argument and object refer to the same location in memory. If you modify the argument's state, you also modify the original object.

Ruby appears to use pass by reference when passing mutable objects. For example, consider the following code:

```
1 def uppercase(value)
2   value.upcase!
3 end
4
5 name = 'William'
6 uppercase(name)
7 puts name               # WILLIAM
```

Here, our method can modify the `name` String through its alias `value`, so it looks like ruby is pass by reference here.

## It's References All The Way Down

At this point, you should recognize the mental model we arrived at in the first two articles of this series. Going back to Understand Variable References and Mutability article, though, you'll recall that ruby's variables don't contain objects; they are merely references to objects. Even if we pass a literal to a method, ruby will first convert that literal to an object, then, internally, create a reference to the object. You can think of such literal references as anonymous — unnamed — references.

If pass by value is employed for immutable objects, but all variables are references, then what exactly is happening when we pass an immutable object? Let's run a short test:

```
1 def print_id number
2   puts "In method object id = #{number.object_id}"
3 end
4
5 value = 33
6 puts "Outside method object id = #{value.object_id}"
7 print_id value
```

This code prints:

```
1 Outside method object id = 67
2 In method object id = 67
```

Quite clearly, `number` and `value` reference the same object despite the object being immutable. We can also see that `value` was not copied. Thus, ruby is not using pass by value. It appears to be using pass by reference.

This completely blows away half of our mental model. Now what?

Mental models are meant to be refined; when new information comes to light that contradicts or enhances the old model in some way, the model needs to adapt. Quite clearly, we have shown that ruby appears to be using pass by reference when passing immutable objects. At this point, in fact, it appears to be using pass by reference all of the time.

You may be ready to ask "But, we can't modify immutable objects! Isn't that what pass by reference is all about?" The key here is that pass by reference isn't limited to mutating methods. A non-mutating method can use pass by reference as well, so pass by reference can be used with immutable objects. There may be a reference passed, but the reference isn't a guarantee that the object can be modified.

## Pass By Reference Value

We can leave things like this and just say that ruby is pass by reference, and we wouldn't be wrong. There's nothing wrong with this conclusion. Many people will tell you that ruby is exclusively pass by reference.

However, assignment — we're back to that again — throws in a bit of a monkey wrench. In a pure pass by reference language, assignment would be a mutating operation. In ruby, it isn't, and the reason for this was discussed earlier; ruby variables and constants aren't objects, but are references to objects. Assignment merely changes which object is bound to a particular variable.

While we can change which object is bound to a variable inside of a method, we can't change the binding of the original arguments. We can change the objects if the objects are mutable, but the references themselves are immutable as far as the method is concerned.

This sounds an awful lot like pass by value. Since pass by value passes copies of arguments into a method, ruby appears to be making copies of the references, then passing those copies to the method. The method can use the references to modify the referenced object, but since the reference itself is a copy, the original reference cannot be changed.

Given all of this, it's not uncommon to just say that ruby is *pass by reference value*, *pass by reference of the value*, or *pass by value of the reference*. It's all a little muddy, but the 3 terms mean essentially the same thing: ruby passes around copies of the references. In short, ruby is neither pass by value nor pass by reference, but instead employs a third strategy that blends the two strategies.

## Final Mental Model

Is that our final answer to the question of whether ruby is pass by reference or pass by value? It's neither? Yes. Well, maybe not entirely; there are actually three answers to the question of what object passing strategy ruby uses:

- **pass by reference value** is probably the most accurate answer, but it's a hard answer to swallow when learning ruby, and isn't particularly helpful when trying to decide what will happen if a method modifies an argument – at least not until you fully understand it.
- **pass by reference** is accurate so long as you account for assignment and immutability.
- Ruby acts like **pass by value** for immutable objects, **pass by reference** for mutable objects is a reasonable answer when learning about ruby, so long as you keep in mind that ruby only *appears* to act like this.

## Wrap-up