Ruby Methods

You'll often have a piece of code that needs to be executed many times in a program. Instead of writing that piece of code over and over, there's a feature in most programming languages called a procedure, which allows you to extract the common code to one place. In Ruby, we call it a **method**. Before we can use a method, we must first define it with the reserved word `def`. After the `def` we give our method a name. At the end of our method definition, we use the reserved word `end` to denote its completion.

Many languages distinguish between functions, which have no associated object, and methods, which are invoked on a receiver object. Because Ruby is a purely object-oriented language, all methods are true methods and are associated with at least one object. Basically, because everything in Ruby is an object, Ruby only has methods, not functions. With that established, we know that when we're talking about Ruby, "methods" and "functions" refer to the same thing.

Ruby's Built-in Methods

One of Ruby's great advantages for new programmers is its large number of built-in methods. You've been using many of them already, probably without even realizing it. Over the course of your learning so far, you have modified strings and other objects in various ways. For example, the `#times` and `#upto` loops that you learned about in the Loops lesson are both methods that are included as part of Ruby's `Integer` class.

If you're wondering about all of the pound signs (#), they're just the convention for writing out Ruby instance methods. We can use them to write out the full name of an instance method, e.g., `Integer#upto`, or just the method name, e.g., `#upto`, depending on the context. Note that in the development world, you shouldn't call these "hashtags". If you want to be super awesome, though, you can call them "octothorpes". That word is totally trending.

Methods are typically called by adding `.method_name` after an instance of the object that contains that method.

```
"anything".reverse
```

In this case, `#reverse` is a built-in method for String objects.

However, there are also some built-in methods that Ruby makes globally accessible, such as `print` and `puts`. These methods are called with just their name and any arguments. (If you're super curious, these methods are made globally available by the Kernel module through the Object class, but that's far more than you need to know right now.)

```
puts "anything" #=> anything
```

It's worth noting that in most languages, arguments are passed to methods by wrapping them in parentheses `()`. In Ruby, however, the parentheses are *generally* optional. We could rewrite the above code as `puts("anything")`, which Ruby would interpret in the same way.

Creating a Method

You can create your own custom methods in Ruby using the following syntax:

```
def my_name
  "Joe Smith"
end
puts my_name    #=> "Joe Smith"
```

`// def` is a built-in keyword that tells Ruby that this is the start of a method definition

`// my_name` is the name of your new method.

`// "Joe Smith"` is the code inside the method body. All of the logic for your method is contained in the indented lines of the method body. This particular method returns a string when the method is called.

`// end` is a built-in keyword that tells Ruby that this is the end of the method definition.

Method Names

Your method names can use numbers, capital letters, lowercase letters, and the special characters _, `?`, `!`, and `=`. Just like with variable names in Ruby, the convention for a method name with multiple words is to use **snake_case**, separating words with underscores.

Here are some things you are not allowed to do with your method names:

- You cannot name your method one of Ruby's approximately 40 reserved words, such as `end`, `while`, or `for`.
- You cannot use any symbols other than _, `?`, `!`, and `=`.
- You cannot use `?`, `!`, or `=` anywhere other than at the end of the name.
- You cannot begin a method name with a number.

Here are some examples of valid and invalid method names:

```
method_name      # valid
_name_of_method  # valid
1_method_name    # invalid
method_27        # valid
method?_name     # invalid
method_name!     # valid
begin            # invalid (Ruby reserved word)
begin_count      # valid
```

Parameters and Arguments

Of course, not all methods are as simplistic as the `my_name` example method above. After all, what good are methods if you can't interact with them? When you want to return something other than a fixed result, you need to give your methods parameters. **Parameters** are variables that your method will receive when it is called. You can have more meaningful and useful interactions with your methods by using parameters to make them more versatile.

```
def greet(name)
  "Hello, " + name + "!"
end
puts greet("John") #=> Hello, John!
```

In this example, `name` is a parameter that the `greet` method uses to return a more specific greeting. The method was called with the argument `"John"`, which returns the string "Hello John!"

If you're wondering what the differences are between an argument and a parameter, **parameters** act as placeholder variables in the template of your method, whereas **arguments** are the actual variables that get passed to the method when it is called. Thus, in the example above, `name` is a parameter and `"John"` is an argument. The two terms are commonly used interchangeably, though, so don't worry too much about this distinction.

Default Parameters

What if you don't always want to provide arguments for each parameter that your method accepts? That's where default parameters can be useful. Going back to our simple example above, what happens if we don't know the person's name? We can change our `greet` method to use a default `name` of "stranger":

```
def greet(name = "stranger")
  "Hello, " + name + "!"
end
puts greet("Jane") #=> Hello, Jane!
puts greet #=> Hello, stranger!
```

What Methods Return

An important detail that a programmer must learn is understanding what your methods **return**. Having a good understanding of what your methods are returning is an important part of debugging your code when things don't behave as expected.

How do we tell our methods what to return? Let's revisit our `my_name` example method:

```
def my_name
  "Joe Smith"
end

puts my_name #=> "Joe Smith"
```

Our `my_name` method returns "Joe Smith". This may seem obvious because it's the only thing inside the method. In most languages, however, such a method would not return anything because it does not have an **explicit return** statement, which is a statement that starts with the `return` keyword. The above example could just as easily be written with an explicit return:

```
def my_name
  return "Joe Smith"
end

puts my_name #=> "Joe Smith"
```

Ruby is one of the few languages that offers **implicit return** for methods, which means that a Ruby method will return the last expression that was evaluated even without the `return` keyword. The last expression that was evaluated may or may not be the last line in the code, as you can see in the following example:

```
def even_odd(number)
  if number % 2 == 0
    "That is an even number."
  else
    "That is an odd number."
  end
end

puts even_odd(16) #=>  That is an even number.
puts even_odd(17) #=>  That is an odd number.
```

Here, the method's return is dependent on the result of the `if` condition. If the argument is an even number, the expression inside the `else` statement never gets evaluated, so the `even_odd` method returns `"That is an even number."`

Even though Ruby offers the ease of using implicit returns, explicit returns still have a place in Ruby code. An explicit return (using the keyword `return`) essentially tells Ruby, "This is the last expression you should evaluate." This example shows how using `return` stops the method from continuing:

```
def my_name
  return "Joe Smith"
  "Jane Doe"
end

puts my_name #=> "Joe Smith"
```

For example, an explicit return can be useful when you want to write a method that checks for input errors before continuing.

```
def even_odd(number)
  unless number.is_a? Numeric
    return "A number was not entered."
  end

  if number % 2 == 0
    "That is an even number."
  else
    "That is an odd number."
  end
end

puts even_odd(20) #=>  That is an even number.
puts even_odd("Ruby") #=>  A number was not entered.
```

Now, try removing the explicit `return` from the method above. Does the method return what you expected?

Difference Between `puts` and `return`

A common source of confusion for new programmers is the difference between `puts` and `return`.

- `puts` is a method that prints whatever argument you pass it to the console.
- `return` is the final output of a method that you can use in other places throughout your code.

For example, we can write a method that calculates the square of a number and then `puts` the output to the console.

```
def puts_squared(number)
  puts number * number
end
```

This method only prints the value that it calculated to the console, but it doesn't return that value. If we then write `x = puts_squared(20)`, the method will print `400` in the console, but the variable `x` will be assigned a value of `nil`. (If you need a refresher on this, go back and review the Ruby Input and Output lesson.)

Now, let's write the same method but with an implicit return instead of `puts`. (Using an explicit return will act exactly the same in this example.)

```
def return_squared(number)
  number * number
end
```

When we run the `return_squared` method, it won't print any output to the console. Instead, it will `return` the result in a way that allows it to be used in the rest of your code. We can save the output of running this method in a variable (`x` in the code below) and use that variable in a variety of ways. If we would still like to see the result of the method in the console, we can `puts` that variable to the console using string interpolation.

```
x = return_squared(20) #=> 400
y = 100
sum = x + y #=> 500

puts "The sum of #{x} and #{y} is #{sum}."
#=> The sum of 400 and 100 is 500.
```

Chaining Methods

One of the magical properties of methods that allows you to write very concise code is being able to **chain methods** together. This can be done using Ruby's built-in methods or with methods that you create.

```
phrase = ["be", "to", "not", "or", "be", "to"]

puts phrase.reverse.join(" ").capitalize
#=> "To be or not to be"
```

Chaining methods together like this effectively has each method call build off of the outcome of the previous method in the chain. The process that takes place essentially produces the following steps:

```
["be", "to", "not", "or", "be", "to"].reverse
= ["to", "be", "or", "not", "to", "be"].join(" ")
= "to be or not to be".capitalize
= "To be or not to be"
```

Predicate Methods

You will sometimes encounter built-in Ruby methods that have a question mark (**`?`**) at the end of their name, such as **`even?`**, **`odd?`**, or **`between?`**. These are all **predicate** methods, which is a naming convention that Ruby uses for methods that return a Boolean, that is, they return either **`true`** or **`false`**.

```
puts 5.even?  #=> false
puts 6.even?  #=> true
puts 17.odd?  #=> true

puts 12.between?(10, 15)  #=> true
```

You can also create your own method with a **`?`** at the end of its name to indicate that it returns a Boolean. Ruby doesn't enforce this naming convention, but you will thank yourself later for following this guideline.

Bang Methods

Observe the example below:

```
whisper = "HELLO EVERYBODY"

puts whisper.downcase #=> "hello everybody"
puts whisper #=> "HELLO EVERYBODY"
```

What gives? I thought we downcased that thing! So why was it back to all uppercase when we called it again?

When we call a method on an object, such as our `whisper` string above, it does not modify the original value of that object. A general rule in programming is that you do not want your methods to overwrite the objects that you call them on. This protects you from irreversibly overwriting your data by accident. You *are* able to overwrite your data by explicitly re-assigning a variable (such as `whisper = whisper.downcase)`. Another way to do this type of reassignment is with **bang methods**, which are denoted with an exclamation mark (**`!`**) at the end of the method name.

By adding a `!` to the end of your method, you indicate that this method performs its action and simultaneously overwrites the value of the original object with the result.

```
puts whisper.downcase! #=> "hello everybody"
puts whisper #=> "hello everybody"
```

Writing `whisper.downcase!` is the equivalent of writing

```
whisper = whisper.downcase.
```