Methods and Arguments

Understanding Arguments

Imagine needing to build a method that greets a person. We could code something like this:

```
def greeting
  puts "Hi, Ruby programmer!"
end
```

This method, when called, will print out to the terminal, the string `"Hi, Ruby programmer!"`. Try it out, open an IRB session by running `irb` from your command line. Once you're in your IRB shell, paste in the code:

```
def greeting
  puts "Hi, Ruby programmer!"
end
// ♥ irb
2.2.1 :001 > def greeting
2.2.1 :002?>   puts "Hi, Ruby programmer!"
2.2.1 :003?>   end
 => :greeting
```

You've now defined the method. Notice that it did not execute. Type the following into IRB to execute your method: `greeting`.

```
2.2.1 :004 > greeting
```

You should see:

```
Hi, Ruby programmer!
 => nil
```

As amazing as this method is, it's still pretty literal. It hard-codes, or directly specifies, the name of the person we are greeting as `"Ruby programmer"`. If we wanted to build a method that can greet anyone, even Python programmers, we'd have to re-implement the majority of the original logic from `greeting`:

```
def greeting_python
  puts "Hello, Python programmer!"
end
```

Notice the only things that changed are the method name and the language name `"Python"` in the body of the method. It's as though that information should be specifiable or configurable when you call the method, otherwise we'd have to build every permutation of the method. In other words, we'd have to re-write the method for every single person we want to greet. We want our method to be more dynamic, more abstract, more re-usable. It should maintain the elements that will always be the same, no matter who we greet, and allow us to change, or swap out, the name of the person we are greeting. This is dynamic, as opposed to "hard-coded".

Good news, that's exactly what method arguments (also called parameters) are for:

```
def greeting(name)
  puts "Hello, #{name}!"
end
```

Above, we define our method to take in an argument by following the method name with parentheses enclosing a variable name: `greeting(name)`.

Then, we use **string interpolation** inside the method body to `puts` out a greeting using whatever `name`was passed into the argument when the method is called. String interpolation allows users to use a Ruby variable to render a value inside of a string. In other words, if we have a variable, `name`, that points to a value of `"Sophie"`, string interpolation will let us use that `name` variable inside a string to render, or `puts` out, a string that contains the word `"Sophie"`.

To interpolate a variable into a string, wrap that variable name inside curly braces, preceded by a pound sign: `#{variable_name}`.

Let's call our method and see it in action:

```
greeting("Sophie")
# > Hello, Sophie!
```

Let's take a closer look at how to add arguments to our methods.

Defining Method Arguments

To add arguments to a method, you specify them in the method signature—the line that starts with `def`. Simply add parentheses after the name of the method and create a placeholder name for your argument.

For example, if I want to write a method called `greeting_a_person` that accepts an argument of a person's name, I would do it like this:

```
    #method name       #argument
def greeting_a_person(name)
  "Hello #{name}"
end
```

Arguments create new local variables that can be used within the method. When you name an argument, you are defining what bare word you want to use to access that data, just like when you create a variable. Arguments follow the same rules as local variables: they can be any word that starts with a lowercase letter and they should be as descriptive of the data as possible.

In our `#greeting` method example, we are saying: When you call the `#greeting` method with an argument of `"Sophie"`, set a variable `name` equal to the value of `"Sophie"`.

Defining Methods with Multiple Arguments

You can define a method to accept as many arguments as you want. Let's try creating a method that accepts two arguments: a person's name and their programming language of choice.

```
  # method name       first_argument, second_argument
def greeting_programmer(name, language)
  puts "Hello, #{name}. We heard you are a great #{language} programmer."
end

greeting_programmer("Sophie", "Ruby")
# > Hello, Sophie. We heard you are a great Ruby programmer.

greeting_programmer("Steven", "Elixir")
# > Hello, Steven. We heard you are a great Elixir programmer.
```

To accept multiple arguments, simply separate the bare words in the argument list with commas.

Required Arguments

Once you define arguments for a method, they become required when you invoke or call the method. If you define a method that accepts a singular argument, when you call that method, you must supply a value for that argument, otherwise, you get an `ArgumentError`. Here's an example:

```
def greeting(name)
  puts "Hello, #{name}!"
end

greeting # I explicitly call the method without a value for the argument `name`
# > ArgumentError: wrong number of arguments (0 for 1)
```

In Ruby, all arguments are required when you invoke the method. You can't define a method to accept an argument and call the method without that argument. Additionally, a method defined to accept one argument will raise an error if called with more than one argument.

```
def greeting(name)
  puts "Hello, #{name}!"
end

greeting("Sophie", "Ruby") # The method accepts 1 argument and I supplied 2.
# > ArgumentError: wrong number of arguments (2 for 1)
```

By default, all arguments defined in a method are required in order to correctly invoke (or "call", or "execute") that method.

Using Arguments in Methods

Now that we know how to define a method with arguments, let's take a closer look at using those arguments, that data, within the method. Once again, our greeting method;

```
def greeting(name)
  puts "Hello, #{name}"
end
```

When we define a method with arguments we are defining a bareword that we can use to reference the actual value supplied to the method upon invocation. We built a method that will greet a specified person. In order to write code in our method to actually greet any given person, we need a placeholder—a way to refer to a generic person's name. This is an argument.

When we build that method we might ask ourselves, "who is this method designed to greet?". The answer is "anyone, it doesn't matter." That's what makes the method abstract, the detail of who it greets is hidden until the method is actually invoked: `greeting("Sophie")`. Only then do we know that the method greets Sophie. The value of `name` is only supplied upon invocation.

The bareword, in this case `name`, that we use as the argument's name in the method signature becomes a local variable within the method. Through that variable we can reference the value of the argument supplied at invocation.

With the code above, when we say: `greeting("Sophie")`, the value of the argument `name` is `"Sophie"`. During the particular runtime invoked by `greeting("Sophie")`, any reference to `name` will have the value of `"Sophie"`, allowing the method to behave as intended.

Similarly, when we say: `greeting("Ann")`, the value of the argument `name` is `"Ann"`.

Method arguments simply create local variables for you to refer to the value used when the method is actually invoked.

A Note on Calling Methods

In the above examples, we're calling methods with parentheses, e.g., `greeting('Sophie')`. But you can also omit the parentheses: `greeting 'Sophie'`.

When a method takes an argument, omitting the parentheses is generally considered bad style, as it's a bit more difficult to understand what's going on. However, when you want to call a method without any arguments — e.g.,

```
def say_hi
  puts "Hi!"
end

say_hi
```

omitting the parentheses helps to clear things up. You might also see some Domain Specific Languages (DSLs) that prefer to omit parentheses. You've probably already seen a little bit of RSpec's DSL, for example:

```
describe "MyRubyThing" do
  it "runs" do
    # test here
  end
end
```

`describe` and `it` are just methods — the above could have been written

```
describe("MyRubyThing") do
  it("runs") do
    # test here
  end
end
```

but I think you'll agree that it looks nicer (and is easier to read) without the parentheses.