AJ Natzic

Professor Woodring

CIS 343

9 December 2018

The Ruby Programming Language

According to Ruby’s Official Documentation, and the creator of Ruby himself, Ruby was born on February 24th, 1993. Ruby, created by Yukihiro Matsumoto, was made to be a language that maximizes productivity and minimalizes frustrations or confusion. While this was a core focus of Ruby, Matsumoto also wanted a purely object-oriented programming language.

“I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising.

I knew Python then. But I didn't like it, because I didn't think it was a true object-oriented language---OO features appeared to be add-on to the language. As a language manic and OO fan for 15 years, I really wanted a genuine object-oriented, easy-to-use scripting language. I looked for, but couldn't find one.

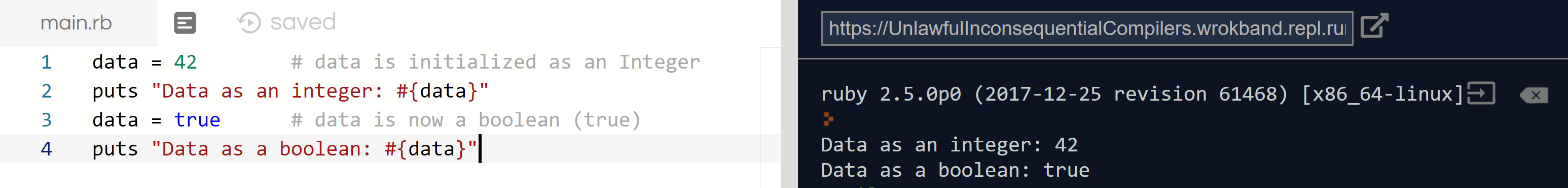
So, I decided to make it. It took several months to make the interpreter run. I put it the features I love to have in my language, such as iterators, exception handling, garbage collection.

Then, I reorganized the features of Perl into a class library, and implemented them. I posted Ruby 0.95 to the Japanese domestic newsgroups in Dec. 1995.

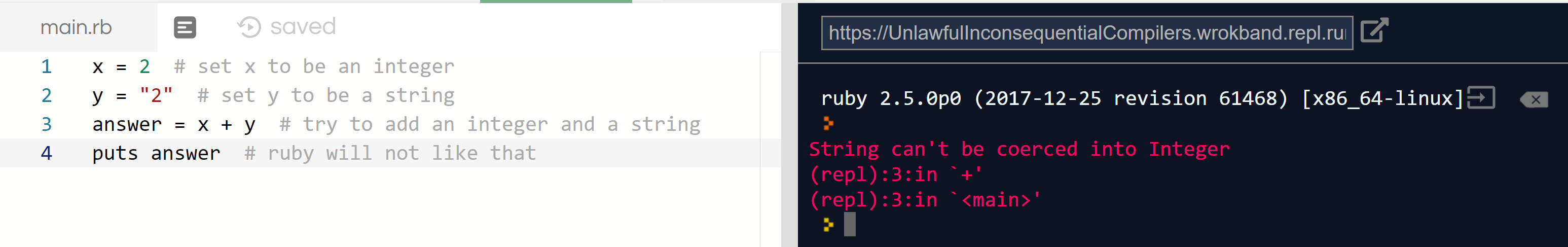
Since then, highly active mailing lists have been established and web pages formed.” (Matsumoto)

Ruby is Object-Oriented, which means that everything in Ruby is an object. Matsumoto goes on to describe that as he developed Ruby more, he blended parts of his favorite languages such as Perl, Smalltalk, Eiffel, Ada, Lisp etc. to create a language that balanced functional programming with imperative programming. Because of this, Ruby is often described as a Multi-paradigm language. So, while Ruby is considered mostly Object-Oriented, it can also be considered imperative, functional and/or reflective. Since Ruby can be defined in many ways, Ruby is used in many different domains. Ruby is good for quick development of programs, dynamic programs, web applications as well as scripts. It is great for doing things that require ambiguity, however it should not be used for things that require costly operations. Those types of problems are better left for a faster language such as C.

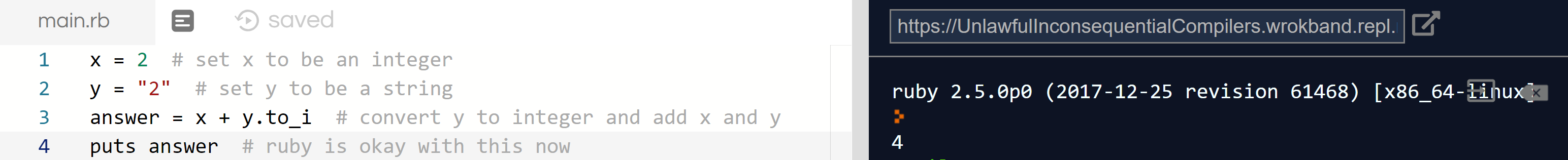
Ruby has several data types; all these data types are based on classes. These data types include Booleans, symbols (used to represent other objects), numbers (float, integer, rational numbers etc.), strings, arrays and hashes. (Bodnar) Ruby is dynamically typed, which means much like Python different data types can be assigned to variables “on-the-fly” because type checking is done at runtime, not compile time. See below code snippet to see how I assigned the value “data” to integer 42, then changed it to a Boolean in Ruby:



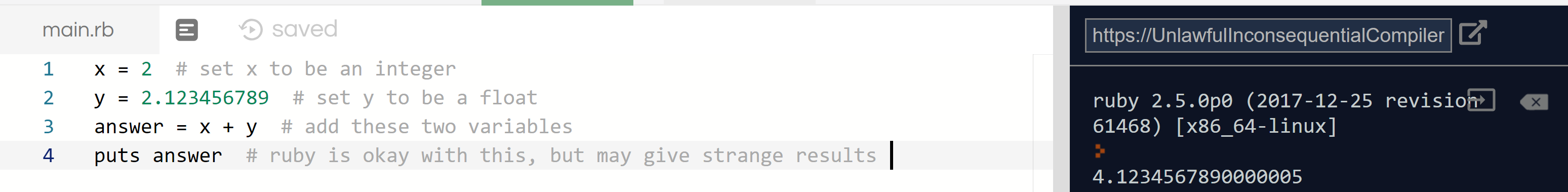
In the above example we can also see that Ruby is using implicitly typed variables. This means that I did not define that data was an integer, or a Boolean when I created it. Ruby inferred these variable types based on their values. This can greatly improve writability since you do not need to explicitly state what your variable types are, and you can change them later if you need to. However, this can reduce readability since it may not be clear to the reader what type Ruby will assign to the data, or what methods you can run on it later. Ruby is also strongly typed, which means that it will only run operations on data types that it knows it will be able to do. So, for example, it will not be able to add a string and an integer together implicitly:



In order to do this operation, we will have to explicitly declare that we want y to be an integer in this case. This is like parameter casting in Java and other programming languages:



Although some data types in Ruby require explicit declarations for operations to be performed, some do not, and coercion will occur:



As you can see, ruby automatically converts ‘x’ to a float so that it can add to ‘y’ and print out a float in the end. However, this can produce strange results, as seen above, since there can be some incorrect conversion done when converting data types.

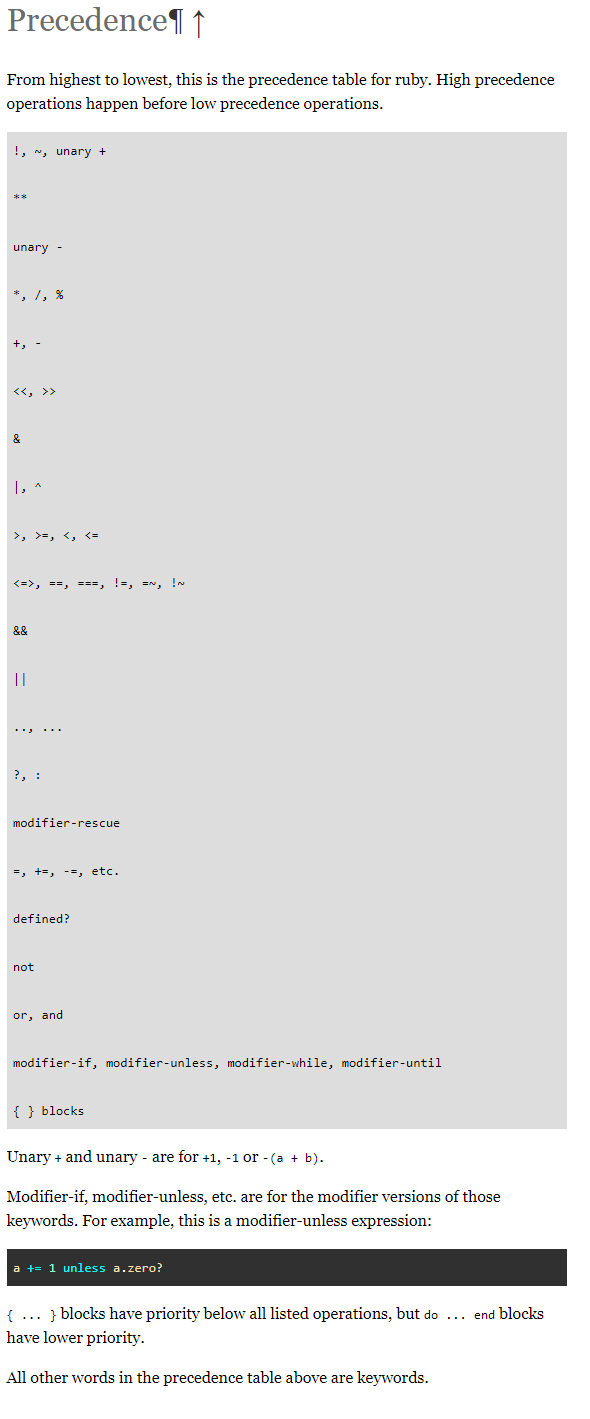
You can see the precedence rules for Ruby in the official ruby documentation. (See Appendix A) The rules seem standard for a programming language, following basic mathematical order of operations. One of the things I found interesting about the precedence rules is that it allows you to use “and” or “&&” for conditionals, either one is acceptable. The same applies for “or” and “||”. Selection constructs in Ruby are defined by the basic “If, then, elsif, else, end” skeleton. An example of this construct in Ruby can be seen on Appendix B. (Savage) It should be noted that you can add a “then” after your conditional if statement if you want to, it will still run like a normal if statement in the example seen in Appendix B. This may be to increase readability and/or writability. The “end” keyword is placed at the end of the conditional. It acts as an end-of-line keyword for the conditional. Much like C, Ruby also has for and while loops. One thing I found interesting about Ruby is the scope rules.

“In Ruby, methods can be called without an explicit receiver and any parentheses, just like local variables. Thus, you can have potential naming conflicts like the one above. [(See Appendix C)] If you have a local variable being initialized and a method call with a same name in the same scope, the local variable will “shadow” the method and take precedence.” (Gjorgjievski)

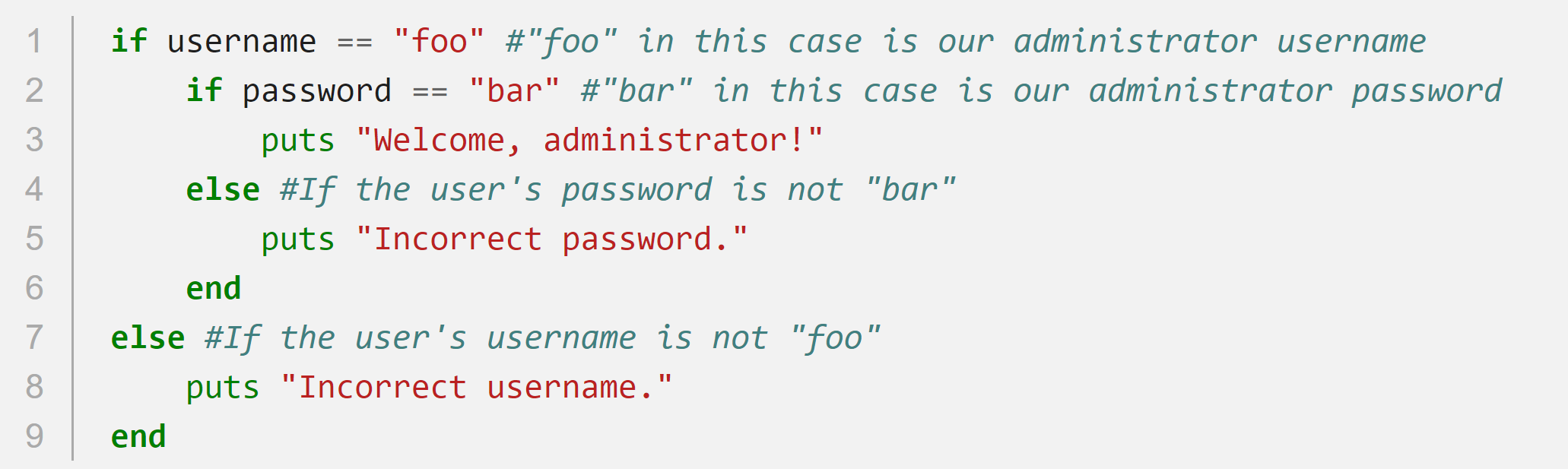
So, you can define a new variable with the same name as a class, call it, and get a different answer that you defined, all at runtime. See Appendix C for an example from Gjorgjievski. Ruby also has exception handling, which means that exceptions can be defined by the user. This makes Ruby a lot safer when developing large projects. Ruby has a similar class syntax to Python. The syntactic units for defining classes are as follows: first use keyword “class” to define the class, then use “def” to include methods and other relevant variables, then use the “end” keyword to terminate the class. Ruby does not support Java-like interfaces, mostly because it does not have a need for them. Since variables and methods are determined at runtime and can be changed at any time, there is no need for abstract interfaces.

Overall, I feel like this is a very interesting language that has a lot of ambiguity. Ruby allows you to use many different keywords in place of others. While this can improve the writability of the language, it can also severely impact readability, for example some people reading your code may not know what the keyword you are using means, they may be more used to something else. I have noticed while researching this language that there are two distinct opinions from most people about Ruby — they either love it or hate it. This may be because Ruby’s biggest strength can also be its biggest weakness. The dynamically and implicitly typed language allows you to do many operations that would be a lot harder to write in other languages. Because of this, naming conflicts and type errors are going to be more common, and readability will be reduced. On Ruby’s official website the tagline says that Ruby is “A programmer’s best friend”, however, that may depend on your previous programming experiences.

Appendix A



Appendix B



1 This is an example of selection constructs in Ruby. It uses if-else statements to determine which code to run. This example was created by Joe Savage and can be found in works cited.

Appendix C



2 An example of naming conflicts due to the scope rules of Ruby. In this example, this can be easily remedied by typing "something()", if you were trying to call the class. This example was provided by Darko Gjorgjievski

Appendix D

Below is my sample Calculator program in Ruby.

# This method ask the user what type of calculation they would like to performdef request\_calculation\_type

puts "\n" + "Do you want to \n[a]dd, \n[s]ubtract, \n[m]ultiply, \n[d]ivide or \n[e]xponent?"

operation\_selection = gets.to\_s.chomp

if operation\_selection == 'a'

return 'add'

elsif operation\_selection == 's'

return 'subtract'

elsif operation\_selection == 'm'

return 'multiply'

elsif operation\_selection == 'd'

return 'divide'

elsif operation\_selection == 'e'

return 'exponent'

else

return 'error'

end

end

# This method returns the result of the calculation

def calculate\_answer(operator, a, b)

if operator == 'add'

a + b

elsif operator == 'subtract'

a - b

elsif operator == 'multiply'

a \* b

elsif operator == 'divide'

a / b

elsif operator == 'exponent'

a \*\* b

end

end

lastAnswer = nil

first\_number = nil

run\_calculator = 1

# runs as long as user lets run\_calculator value be 1

while run\_calculator == 1

current\_calculation = request\_calculation\_type

# throws an exception if user did not input correct operation

if current\_calculation == 'error'

puts "\n" + 'ERROR: Unknown calculation. Type in a, s, m, d or e when prompted'

else

if first\_number.nil? # if there is no first number, ask for one

puts "\n" + "What is the first number you would you like to #{current\_calculation}: "

first\_number = gets.to\_f

end

puts "\n" + "What is the second number you would like to #{current\_calculation}: "

second\_number = gets.to\_f

# throws an exception if user tries to divide by zero

if (current\_calculation == 'divide') && (second\_number == 0)

puts "\n" + 'ERROR: Divide by Zero'

else

answer = if lastAnswer.nil?

calculate\_answer(current\_calculation, first\_number, second\_number)

else

calculate\_answer(current\_calculation, answer, second\_number)

end

puts "\n" + "The answer is #{answer}"

lastAnswer = answer # updates lastAnswer for future calulations

puts "\n" + 'Type 1 to run another calculation or anything else to end: '

run\_calculator = gets.to\_i

end

puts "\n" + 'Ending Calculator' if run\_calculator != 1

end

end

Works Cited

Bodnar, Jan. “Ruby Data Types.” ZetCode, 26 Feb. 2018, zetcode.com/lang/rubytutorial/datatypes/.

Gjorgjievski, Darko. “Understanding Scope in Ruby.” SitePoint, 13 Aug. 2015, www.sitepoint.com/understanding-scope-in-ruby/.

Matsumoto, Yukihiro “Ruby's Frequently Asked Questions.” *Ruby Official Documentation*, ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html.

“Ruby Programming Language.” About Ruby, [www.ruby-lang.org/](http://www.ruby-lang.org/).

Savage, Joe. “Ruby: If Statements.” Dev-HQ, www.dev-hq.net/ruby/5--if-statements.