From JavaScript to Ruby

Ruby is a dynamic, strongly typed programming language derived from Smalltalk with a little bit of Perl mixed in. It shares many things with JavaScript, so in this guide we'll look at Ruby from the perspective of a JavaScript programmer.

Along the way you'll do some coding, so it's in your best interest to type out every example here to get practice.

Running Ruby programs

Unlike JavaScript, Ruby can't run in a web browser. You have to make a Ruby file and then run that Ruby file through the Ruby interpreter. Or you could use the Interactive Ruby program.

Interactive Ruby (irb)

The command `irb` gives you a Ruby console that you can use much like the JavaScript console in your browser. Type valid Ruby commands and see the results.

Running Ruby programs

Save your code in a file with the `.rb` extension and run them through the Ruby interpreter

```
$ ruby hello.rb
```

Printing Output

In JavaScript, you write code like this to print output:

```
console.log("Hello");
```

Or if you were writing text to a web browser's document, you'd do this:

```
document.write("Hello");
```

In Ruby, you use

```
print("Hello")
```

Or, because in Ruby, parentheses are optional when there's no ambiguity, you can just say

```
print "Hello"
```

Pretty simple. If you'd like an automatic line break added to the end of your output, you can use `puts` instead of `print`. This is much more common.

You try it

Create the program `myname.rb` with a line that prints your name. Run the program with

```
$ ruby myname.rb
```

Then add a second line of output to the program that prints any other text you'd like. Run the program again.

Declaring Variables

Variables don't have to be declared before you use them. The main reason for this is that you really can't. Ruby doesn't have a `var` keyword. The only way to declare a variable before you use it is to assign a value to a variable.

In JavaScript you might do this:

```
var number1;
var number2;
var total = 0;
```

In Ruby, you don't have to declare variables before you use them, but it's actually a good idea to do so. However, in Ruby, you declare a variable by assigning it a value, so to declare your variables before you use them, you'd need to do this:

```
number1 = 0;
number2 = 0;
total = 0;
```

Ruby has no equivalent of the `var` keyword. Ruby uses *type inference* to set the data type. Unlike JavaScript, Ruby is **strongly** typed. However, like JavaScript, it is dynamically typed.

Naming Conventions

In JavaScript, naming conventions are pretty arbitrary and mostly up to the developer. In Ruby, naming conventions matter.

Capital Letters Mean Constants!

If you declare a variable with a capital letter, Ruby defines it as a Constant, which is a variable that cannot have its value changed.

- In Ruby, you define objects as constants, so all of your objects will have a capital letter at the beginning.
- Objects use `SnakeCase` when you have multiple words. `DateTime`, not `dateTime`.
- You'll use lower case variable names for variables, functions/methods, and everything else.
- Variables and function names should use underscores for multiple words. That means you do `first_name` and not `firstName` or `FirstName`.
- To differentiate "constants" from Objects, you'll use ALL*CAPITAL*LETTERS for your constant names. Like `MAX_AGE = 99`

That's quite a bit of rules to remember. But as you use them and apply them, they'll actually start to make sense. Consider this example:

```
class Person
  ROLES = ["admin", "user"]
  attr_accessor :first_name, :last_name

  def full_name
    "#{self.first_name} #{self.last_name}
  end
end
```

This example contains a lot of stuff we haven't covered. However, by looking at it, we can tell the following:

- `Person` is an object
- `ROLES` is a constant

- `attr_accessor` , `:first_name` , and `:last_name` and `full_name` are all variables or functions of some type.

These conventions make it easy for every Ruby developer to understand the intentions of every other Ruby developer. Conventions are incredibly important to all Rubyists.

You try it

Write a program called `numbers.rb` that has two variables: `firstNumber` and `secondNumber` . Assign them initial values.

Print out both numbers to the screen, each on its own line.

Prompting for input

In JavaScript, you could use code like this to get input from the user:

```
var name = prompt("What is your name?");
```

In Ruby, you'd do this:

```
puts "What is your name?"
name = gets.chop
```

The first line prints the text to the screen, prompting the user so they know what to do.

The second line waits for input. When the user presses the `Enter` key, then everything they typed, INCLUDING the Enter keypress (the line break character), gets stored in the variable on the left-hand side.

The `chop` method simply "chops off" the last character of the string. The last character is usually the `\n` character generated by the user pressing the ENTER key. So we get rid of that

If the user enters "Homer" and presses the Enter key to make the program continue, then the data that the Ruby program receives is

```
"Homer\n"
```

Using `.chop` takes the last character off, so that only `Homer` is stored in the variable. If you forget the `.chop` , then it's not a big deal, but when you use your variable's value you might end up with an extra line break when you print things out.

You try it

Write a program that prompts for your name. Then print your name back to the screen.

Math

Ruby uses the same symbols as JavaScript for math

Operator	symbol
Addition	+
Subtraction	-
Multiplication	*
Division	-
Modulo	%

Ruby **does not** have the `++` or `--` unary operators. To increment an existing variable, you must use a compound operator like `+=`:

```
age = 12
age += 1
```

Converting to numbers

Here's an addition program in JavaScript that takes in values from the user:

```
var answer;
var number1 = prompt("Enter the first number.');
var number2 = prompt("Enter the second number.");

// Conversion
number1 = Number(number1);
number2 = Number(number2);
```

```
// Math
answer = number1 + number2;

// Output
document.write(answer);
```

In this program, the numbers come in as strings. We convert them to numbers and then add them together.

If we don't convert the numbers, then the values get concatenated. They don't get summed up.

In Ruby, the above program looks like this:

```
answer = 0
puts "What is the first number?"
number1 = gets.chop

puts "What is the second number?"
number2 = gets.chop

# Converting to integers
number1 = number1.to_i
number2 = number2.to_i

# Math
answer = number1 + number2

# Output
puts answer
```

We **have to use** the `.to_i` *method* on a string to convert it to a number.

This is because **Everything in Ruby is an object.**

Even numbers. This'll be important very soon.

Strong Typing

In JavaScript, this works:

```
var age = 42;
var name = "Homer";
```

```
document.write(name + " is " + age + " years old.");
```

JavaScript *knows* you want to print out a `String`, so it **coerces** `age` to be a `String`. This is *weak typing* in action.

Ruby will not allow this. This program will crash:

```
age = 42
name = "Homer"
puts name + " is " + age + " years old."
```

In Ruby, a `String` and a `Fixnum` (Ruby's name for an integer, meaning "fixed number") can't be added together. We have to *cast* one type to another.

One way to do this is by using the `to_s` method of a number. Remember, everything's an object, so even numbers have methods we can call. The `to_s` turns a number into a `String.`

```
age = 42
name = "Homer"
puts name + " is " + age.to_s + " years old."
```

You try it

Create a program that prompts for three numbers. Add the numbers together and print out the result.

Given inputs of `1`, `2`, and `3`, the program output should be this:

```
The total of 1, 2, and 3 is 6.
```

String Interpolation

Embedding numbers in strings is really common. Ruby provides a mechanism to do that. In a double-quoted string, we can use the expression `#{ }` to evaluate Ruby code and automatically convert the result to a string.

In other words, this line:

```
puts name + " is " + age.to_s + " years old."
```

can actually be written like this:

```
puts "#{name} is #{age} years old."
```

See how it removes all of the `+` operators? The variables are "embedded" and "interpolated" into the string. This is incredibly handy and is **the way** that Rubyists build up strings for output.

You try it

Alter your program that adds three numbers so that it uses string interpolation.

Type Checking and Type Casting

Ruby does not have a built-in `isNaN` function that tests to see if a string is numeric. However, if you convert the input string to a number, you can figure it out. Strings like `a` get converted to `0` if you convert them

So, to test to see if a string is numeric you can convert the entry to a number using `.to_i` and then convert it back to a string using `.to_s`. If the converted value is the same as the original value, it's numeric.

```
number = "1"
number.to_i.to_s == number
=> true

number = "a"
number.to_i.to_s == number
=> false
```

There are ways to do this with regular expressions too, but this is the simplest for now.

Here are the common types for simple data

type	description
Fixnum	A fixed integer (32 bit)
Bignum	A fixed integer > 32 bits
Float	A floating point number (a decimal)
String	A group of characters
Symbol	A string used as a label. Like `:admin`

Here are the methods you can use to convert types:

method	description
`.to_i`	Convert to integer. Non-numeric strings get turned to `0`. Works on Strings, Fixnums, and Floats
`.to_f`	Convert to a floating point. Works on Strings, Fixnums, and Floats
`.to_s`	Convert to a String. Works on every object.
`.to_sym`	Convert to a Symbol

Checking The Type

In JavaScript, you use `typeof` to determine the data type:

```
var age = 42

typeof age

"String"
```

Since everything in Ruby is an object, you use the `.class` method on a variable.

```
age = 42
age.class
=> Fixnum
```

You can also use the `is_a?` method:

```
age = 42
age.is_a? String
=> true
```

Control flow

Ruby has `if` and `else` statements just like JavaScript.

Here's a simple `if` statement in JavaScript:

```
var age = 16;
if(age >= 16){
  document.write("You can drive!");
}
```

In Ruby, here's the code:

```
age = 16
if age >= 16
  puts "You can drive."
end
```

However, for simple IF statements, we can use an "if predicate.":

```
age = 16
puts "You can drive" if age >= 16
```

Unless

Ruby also offers an `unless` statement which gets used instead of "if not".

```
puts "You can drive" unless age < 16
```

The `unless` statement can be a normal "block" statement too:

```
unless age < 16
  puts "You can drive."
end
```

You use `unless` when it makes sense from a readability standpoint.

If/else

In JavaScript you have `if/else` statements like this:

```
var age = 16;
if(age >= 16){
  document.write("You can drive!");
}else{
  document.write("Sorry! No driving for you!");
}
```

In Ruby, the syntax is similar to what you might see in pseudocode:

```
age = 16
if age >= 16
  puts "You can drive!"
else
  puts "Sorry! No driving for you!"
end
```

No extra curly braces for scope. The `else` and `end` handle that.

IF/ELSEIF/ELSE

In JavaScript, you simply do `else if`

In Ruby, you have to use `elsif` (without the inner `e`. It's awkward on purpose. Ruby is an OO language - if you're using an `elsif` statement there's probably a better way to code what you're doing.

Functions

In JavaScript you have two ways to make functions:

```
function hello{
  document.write("Hello world!");
}
```

or

```
var hello = function(){
  document.write("Hello world!");
};
```

In Ruby, we use the `def` keyword:

```
def hello
  puts "Hello world"
end
```

In Ruby, if the function takes no arguments, we do not need to use parentheses when we declare it.

In Ruby, if the function takes no arguments, we **do not** need to use parentheses when we **call** the function.

```
  hello
```

Will work just fine. No need to do

```
  hello()
```

This is a very big difference between Ruby and JavaScript.

Return Values and Void functions

Ruby has no void functions. In JavaScript, you use the `return` keyword to return a value, or your function simply becomes a `void` function (a function without a return value.)

Ruby has no such mechanism. The `return` keyword is not necessary - every function implicitly returns a value. That value is **the result of the last expression.**

You Try It!

What's the return value of this function?

```
def add
  1+1
  2+2
  3+3
end
```

How about this one?

```
def add
  a = 1
  b = 2
  c = a + b
  d = 4
  c
end
```

Bonus: What about this one?

```
def add
  a = 1
  b = 2
  return a
  c = a + b
end
```

Arguments

Functions can take arguments. Here's an example:

```
def full_name(first_name, last_name)
  first_name + " " + last_name
end
```

This time, we need the parentheses to define the arguments.

Of course, that's really not how we do it in Ruby. Remember we have string interpolation so we can write the function like this:

```
def full_name(first_name, last_name)
  "#{first_name} #{last_name}"
end
```

Now, when we call this function, we **don't need to use the parentheses! But we really should.

```
puts full_name "brian", "Hogan"
```

You Try It

Write a function called `is_even` that returns a `true` value if the given number is even.

```
puts is_even?(2)
```

should return

```
true
```

Loops

Ruby has loops just like other languages.

Counted loops

In JavaScript, you use `for` loops to count to known value.

```
for(var index = 0; index < 5; index++){
  // some code
}
```

A `for` loop in Ruby works like this:

```
for index in 1..5 do

end
```

But this really isn't that common, as you'll see later.

We can also use the `while` loop for counted loops.

```
index = 0;
while index < 5 do
  # some code
  index = index + 1
end
```

But you will never write code like this. A `while` loop is best used as an uncounted loop.

For counted loops, you'll use code that looks like this:

```
5.times do
  # your code goes here
end
```

This is an example of a Block. But we'll get to that. For right now, know that this is the preferred method for doing any kind of counted loop.

```
puts "How many times should I repeat?"
number = gets.chop

number.times do
  puts "Hello!"
end
```

See? It even works with variables. Because everything is an object, even simple integers.

Uncounted loops

An uncounted loop is a loop that repeats the block of code as long as the condition is true, which **may or may not be** using a counter variable.

In JavaScript, you'd write a `while` loop to keep doing some steps until a condition was false.

```
var keepGoing = true;
while(keepGoing){
  number = prompt("Enter a number or leave blank to stop.");
  if(number == ""){
    keepGoing = false;
  }
  // other code
}
```

In Ruby, it works the same way:

```
keepGoing = true
while(keepGoing) do

  print "What's your number? "
  number = gets.chop
  if number == ""
    keepGoing = false
  end

end
```

You Try It!

Write a program that asks for input until the user enters an empty string. Print out each entered value, and whether the entered value is a string, an odd number, or an even number.

Arrays and Hashes

JavaScript and Ruby arrays are nearly identical.

```
names = ["Homer", "Marge", "Bart"]
```

`names[0]` will give us `Homer`, `names[1]` gives us `Marge`, and so on.

But in Ruby, we can use negative indexes to go from the `right` side of the array.

So, I can always grab the last entry with

`names[-1]`

Arrays in Ruby are true objects, and that means they have methods.

In JavaScript, you can use the `length` property to tell how many entries are in an array. In Ruby, we don't have `properties` - we simply have methods, and we do have a `length` method.

Hashes

Hashes, or dictionaries, are a little different in Ruby.

We declare a hash like this:

```
person = {"name" => "Homer", "age" => 42}
```

We're using strings as the keys in this case. However, you might see something that looks like this:

```
person = {:name => "Homer", :age => 42}
```

This hash is using `Symbols` for the key instead of Strings. A symbol is a special value in Ruby designed for labeling things. It's better than a String because each new String we create is an object, with its own independent memory location, so that gets expensive. Symbols always take up the same amount of memory, no matter how many times we use them.

And if we use Symbols, we can use this syntax instead:

```
person = {name: "Homer", age: 42}
```

which looks just like JavaScript. This syntax **only works when the keys of the Hash are Symbols.** In all other cases, you must use the `=>` symbol, which we call the "hash rocket."

You try it

Create a program called `person.rb.` Use a Hash to create and persist a person. Prompt for a first name, last name, and an age. Store the data in a hash under the keys `first_name`, `last_name` and `age`.

Store the `person` hash in an array you create called `people`.

Prompt the user if they'd like to enter another person. If they say yes, repeat the process. If they say no, print out the number of people they entered by using the `.length` method of the `people` array. Then exit the program.

Blocks

Blocks are one of the most common features of Ruby. Blocks are often referred to as "closures" or "lambdas" in other languages. Blocks allow us to take bits of code and pass it along to a method as if it were another parameter. The code we send is then executed in the context of that method.

That's a little hard to understand, but here's the gist of it: Remember earlier you learned that you can call the `times` method on an integer? We did this code:

```
5.times do
  puts "Hello"
end
```

This is actually an example of a block. The code `puts "Hello"` is passed as an argument to the `times` method, and executed. The `times` method, behind the scenes, is really just doing a `for` loop. If you look at the implementation (which is actually written in the C programming language) you'll see this:

```
    if (FIXNUM_P(num)) {
        long i, end;

        end = FIX2LONG(num);
        for (i=0; i<end; i++) {
            rb_yield(LONG2FIX(i));
        }
    }
```

There it is, a For loop. There's a call to something called `rb_yield` which basically says "Run whatever code is in the block the programmer gave me.

So, when you do

```
5.times do
  puts "Hello"
end
```

It really is the equivalent of doing a for loop with the same thing. But it's more powerful.

Here's why:

Managing Collections with Enumerable

Ruby "collections", like Arrays and Hashes, all share special methods for working with the items in a collection.

Let's say we had an array of usernames and we wanted to print out each one. In JavaScript, we'd write code like this:

```
var names = ["hsimpson", "bsimpson", "cmburns"];
for(var index = 0, size = names.length; index < size; index++){
  console.log(names[i]);
}
```

That works fine. However, it's just a little silly to have to write a loop just to count over the elements of an array. The array is an object. It *knows* how many elements it has, because it has a `.length` property we can call.

So in Ruby, we can use a method called `each` which accepts a block.

The code for this example in Ruby is simply

```
names = ["hsimpson", "bsimpson", "cmburns"]
names.each do |name|
  puts name
end
```

The `name` variable between the pipes is the "yielded" value. In other words, it's the way you "get" at the object your block interacts with. In our case, it's the entry in the array at the current index.

See, we never have to handle the index ourselves. It's completely hidden away from us so we don't have to think about it.

The `each` method works on Hashes too, except that a Hash has both keys *and* values. So we can get at those quite easily:

```
person = {name: "Homer", age: 42}
person.each do |key, value|
  puts key
  puts value
end
```

This time, both the `key` and the `value` are yielded back. The method you are calling determines what you can access. Some methods don't yield a value back to you at all, while others yield back entire objects.

Map / collect

Ruby collections provide other powerful features like `map` which transforms a collection.

For example, if you had an array of usernames that you needed to turn into an array of email addresses, you might iterate over the array, take the current value of the array and concatenate the email domain onto the end, and store that value in a new array. Sounds like a lot of work.

But in Ruby you can do this:

```
names = ["hsimpson", "bsimpson", "cmburns"]
emails = names.map do |name|
  name + "@springfieldnuclear.com"
end
```

In this case, the expression within the block **becomes** the entry in the new array. The `map` method takes care of assigning the evaluated expression of your block to the index of the new array, so you don't have to.

And if you replace the `do` and the `end` with curly braces, you can put this all on one line:

```
emails = names.map {|name| name + "@springfieldnuclear.com"}
```

This is the way Rubyists write single-line blocks.

Your turn

Take your `person.rb` program and modify it so that when completed, it prints out all of the people in the following format:

```
Name: Homer Simpson
Age: 24

Name: Bart Simpson
Age: 10
```

Blocks for Readability

In the Ruby on Rails framework, you declare database tables like this:

```
create_table :users do |t|
  t.string :username
  t.string :email
  t.string :crypted_password
  t.text :bio
end
```

Don't worry about how this works right now. Just look at how clear blocks make the process. You can almost clearly see that this creates a table called "users" with fields for the username, email, a crypted password, and a biography.

Ruby developers use blocks heavily because it results in more readable code. You will use them a ton.

The closest thing JavaScript has to a block is a callback, like this:

```
element.addEventListener(function(event){
  // this code gets called when the event fires
});
```

jQuery and other libraries provide similar features like `map` and `each` by using callbacks. JavaScript's future editions will get native support.

Wrapping Up

That does it for the basics. From here you should be able to write very simple procedural programs using Ruby.