Review: Arithmetic Lab

- Due No Due Date
- Points 1
- Submitting a website url

 (https://github.com/learn-co-curriculum/phase-1-arithmetic-lab)  (https://github.com/learn-co-curriculum/phase-1-arithmetic-lab/issues/new)

Learning Goals

- Recognize the limitations of math in JavaScript
- Employ operators to perform arithmetic and assign values to variables
- Explain what `NaN` is
- Use built-in objects like `Math` and `Number` to accomplish complex tasks

Introduction

We're going to discuss a number of the common operators and objects we'll use to perform arithmetic operations in JavaScript.

In the browser's JavaScript console, we can test out all of the examples in this lesson. Remember that we can't redeclare variables previously declared with `const` or `let`, so the page may have to be refreshed (which wipes away all declared variables) or different variable names can be chosen than those in the examples.

Getting Started

If you haven't already, fork and clone this lab into your local environment. Remember to **fork** a copy into your GitHub account first, then **clone** from that copy. Navigate into its directory in the terminal, then run `code .` to open the files in Visual Studio Code.

Next, run `npm install` to install the dependencies then run the test suite with the `npm test` command.

Recognize the Limitations of Math in JavaScript

Math is awesome! JavaScript has only a single, all-encompassing `number` type. While other languages might have distinct types for integers, decimals, and the like, JavaScript represents everything as a double-precision floating-point number, or *float*. This imposes some interesting technical limitations on the precision of the arithmetic we can perform with JavaScript. For example:

```
0.1 * 0.1;
//=> 0.010000000000000002

0.1 + 0.1 + 0.1;
//=> 0.30000000000000004

1 - 0.9;
//=> 0.09999999999999998
```

You shouldn't waste too much time diving into why this happens, but it basically boils down to the language, once again, trying to be too user-friendly. Under the hood, JavaScript stores numbers in binary (base-2) format, as a series of `1` s and `0` s, but it displays numbers in the more human-readable decimal (base-10) format. The problem that the above code snippet highlights is that it's really easy to represent something like `1/10` in decimal (`0.1`) but impossible to do it in binary (`0.0001100110011...`). It's the exact same problem that the decimal system has in trying to represent `1/3` as `0.33333333333...` .

The only time you'd really have to worry about this is if you needed to calculate something to a high degree of precision, like interest payments for a bank. However, for most of our day-to-day arithmetic needs, JavaScript is more than capable.

Employ Operators to Perform Arithmetic and Assign Values to Variables

JavaScript employs a pretty standard set of arithmetic operators.

Arithmetic Operators

`+`

We've used the addition operator to concatenate strings, but it's also used to add numbers together:

```
40 + 2;
//=> 42
```

-

The subtraction operator returns the difference between two numbers:

```
9001 - 9000;
//=> 1
```

*

The multiplication operator returns the product of two numbers:

```
6 * 7;
//=> 42
```

/

The division operator returns the result of the left number divided by the right number:

```
9001 / 42;
//=> 214.3095238095238
```

%

The remainder operator returns the remainder when the left number is divided by the right number:

```
9001 % 42;
//=> 13
```

`**`

The exponentiation operator returns the left number raised to the power of the right number:

```
2 ** 8;
//=> 256
```

Order of Operations

Suppose you ask JavaScript to compute a statement such as this:

```
2 - 2 % 2 + 2 / 2 ** 2 * 2;
```

How will JavaScript know which operators to process first? Will it process them in a left-to-right order? (`2 - 2` , which gives 0, then `0 % 2` , and so on)? Or does it use a different set of rules?

JavaScript evaluates compound arithmetic operations by following the standard **order of operations (https://en.wikipedia.org/wiki/Order_of_operations)** used in basic math. Anything in parentheses has highest priority; exponentiation is second; then multiplication, division, and remainder; and, finally, addition and subtraction, in order from left to right. This is how the JavaScript compiler works. For example:

```
( ) → ** → *  /  % → +  -
```

```
2 - (2 % 2) + (2 / 2 ** 2) * 2;
//=> 3

2 - ((2 % (2 + 2)) / 2 ** 2) * 2;
//=> 1
```

Incrementing and Decrementing

JavaScript also has a pair of operators that we can use to increment and decrement a numerical value stored in a variable.

++

The `++` operator increments the stored number by `1`. If the `++` operator comes after the variable (e.g., `counter++`), the variable's value is *returned first and then incremented*:

```
let counter = 0;
//=> undefined

counter++;
//=> 0

counter;
//=> 1
```

If the `++` operator comes before the variable (e.g., `++counter`), the variable's value is *incremented first and then returned*:

```
let counter = 0;
//=> undefined

++counter;
//=> 1

counter;
//=> 1
```

In both cases, `counter` contains the value `1` after incrementing. The difference is in whether we want the operation to return the original or incremented value.

--

The `--` operator decrements the stored number by `1` and has the same pair of prefix and postfix options as the `++` operator:

```
let counter = 0;
//=> undefined

// Return the current value of 'counter' and then decrement by 1
counter--;
//=> 0

// Check the new value of 'counter'
counter;
//=> -1

// Decrement 'counter' and then return the new value
--counter;
//=> -2

// Check the new value of 'counter'
counter;
//=> -2
```

Assignment operators

JavaScript has a number of operators for assigning a value to a variable. We've already used the most basic, `=`, but we can also couple it with an arithmetic operator to perform an operation *and* assign the value of the operation:

```
let counter = 0;
//=> undefined

counter += 10;
//=> 10

counter -= 2;
//=> 8
```

```
counter *= 4;
//=> 32

counter /= 2;
//=> 16

counter %= 6;
//=> 4

counter **= 3;
//=> 64
```

Explain What `NaN` Is

JavaScript tries to return a value for every operation, but sometimes we'll ask it to calculate the incalculable. For example, imagine that one of the lines of code in our program increments the value of a `counter` by `1`. However, something broke in a different part of the program, and `counter` is currently `undefined`. When the JavaScript engine reaches the incrementing line, what happens?

```
counter++;
//=> NaN
```

The JavaScript engine can't add `1` to `undefined`, so it tells us the result is **Not a Number** — `NaN`.

> ***Top Tip***: Much like `undefined`, you should never assign `NaN` as the value of a variable and instead let it be a signal that some weird maths are happening in your code.

Use built-in objects like `Math` and `Number` to accomplish complex tasks

To satisfy most of our math needs, JavaScript provides several built-in objects that we can reference anywhere in JavaScript code, including `Number` and `Math`. With these objects, we can perform complex tasks like generating random numbers.

Number

The `Number` object comes with a collection of handy methods that we can use for checking and converting numbers in JavaScript.

Number.isInteger()

Checks whether the provided argument is an integer:

```
Number.isInteger(42);
//=> true

Number.isInteger(0.42);
//=> false
```

Number.isFinite()

Checks whether the provided argument is finite:

```
Number.isFinite(9001);
//=> true

Number.isFinite(Infinity);
//=> false
```

Number.isNaN()

Checks whether the provided argument is `NaN` :

```
Number.isNaN(10);
//=> false

Number.isNaN(undefined);
```

```
//=> false

Number.isNaN(NaN);
//=> true
```

Number.parseInt()

Accepts a string as its first argument and parses it as an integer. The second argument is the base that should be used in parsing (e.g., `2` for binary or `10` for decimal). For example, `100` is `100` in decimal but `4` in binary:

```
Number.parseInt('100', 10);
//=> 100

Number.parseInt('100', 2);
//=> 4
```

The second argument is optional, but you should always provide it to avoid confusion.

Number.parseFloat()

`Number.parseFloat()` only accepts a single argument, the string that should be parsed into a floating-point number:

```
Number.parseFloat('3.14159');
//=> 3.14159
```

Math

The `Math` object contains some properties representing common mathematical values, such as `Math.PI` and `Math.E`, as well as a number of methods for performing useful calculations.

Math.ceil() / Math.floor() / Math.round()

JavaScript provides three methods for rounding numbers. `Math.ceil()` rounds the number *up*, `Math.floor()` rounds the number *down*, and `Math.round()` rounds the number either up or down, whichever is nearest:

```
Math.ceil(0.5);
//=> 1

Math.floor(0.5);
//=> 0

Math.round(0.5);
//=> 1

Math.round(0.49);
//=> 0
```

Math.max() / Math.min()

These two methods accept a number of arguments and return the lowest and highest constituent, respectively:

```
Math.max(1, 2, 3, 4, 5);
//=> 5

Math.min(1, 2, 3, 4, 5);
//=> 1
```

Math.random()

This method generates a random number between `0` (inclusive) and `1` (exclusive):

```
Math.random();
//=> 0.4495507082209371
```

In combination with some simple arithmetic and one of the rounding methods, we can generate random integers within a specific range. For example, to generate a random integer between `1` and `10`:

```
Math.floor(Math.random() * 10) + 1;
//=> 8

Math.floor(Math.random() * 10) + 1;
//=> 1

Math.floor(Math.random() * 10) + 1;
//=> 6
```

`Math.random()` returns a number between `0` and `0.999...`, which we multiply by `10` to give us a number between `0` and `9.999...`. We then pass that number to `Math.floor()`, which returns an integer between `0` and `9`. That's one less than the desired range (`1` to `10`), so we add one at the end of the equation. Try it out in the JS console!

Assignment

There are four challenges we need you to solve. Code your solution in `index.js`. We'll provide some brief instructions here, but you should really rely on the test failure messages to guide your code.

Instructions

1. Create a variable called `multiply` set to an equation that will multiply the variables `num1` and `num2`; the result of the multiplication should be `62`.
2. Create a variable called `random` that will generate a random integer greater than 0.
3. Create a variable called `mod` set to an equation that will calculate the remainder of dividing variable `num3` by `num4`; the remainder should be `4`.
4. Create a variable called `max` that finds the highest number in a set; the value returned should be `20`.

After you have all the tests passing, remember to commit and push your changes up to GitHub, then submit your work to Canvas using CodeGrade. If you need a reminder, go back to the **Completing and Submitting Assignments with CodeGrade (https://github.com/learn-co-**

curriculum/phase-1-completing-assignments-with-codegrade) lesson to review the process.

Resources

- MDN
  - Basic math in JavaScript (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Math)
  - Arithmetic operators (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators)
  - Operator precedence (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)
  - Assignment operators (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators)
  - `NaN` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN)
  - `Number` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number)
  - `Math` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)
- 2ality — How numbers are encoded in JavaScript (http://2ality.com/2012/04/number-encoding.html)
- Order of Operations (https://en.wikipedia.org/wiki/Order_of_operations)