20 Minute Methods

Learning JavaScript
One Bite At A Time

30 Methods / 12 Days / 20 Minutes A Day

by Travis Rodgers

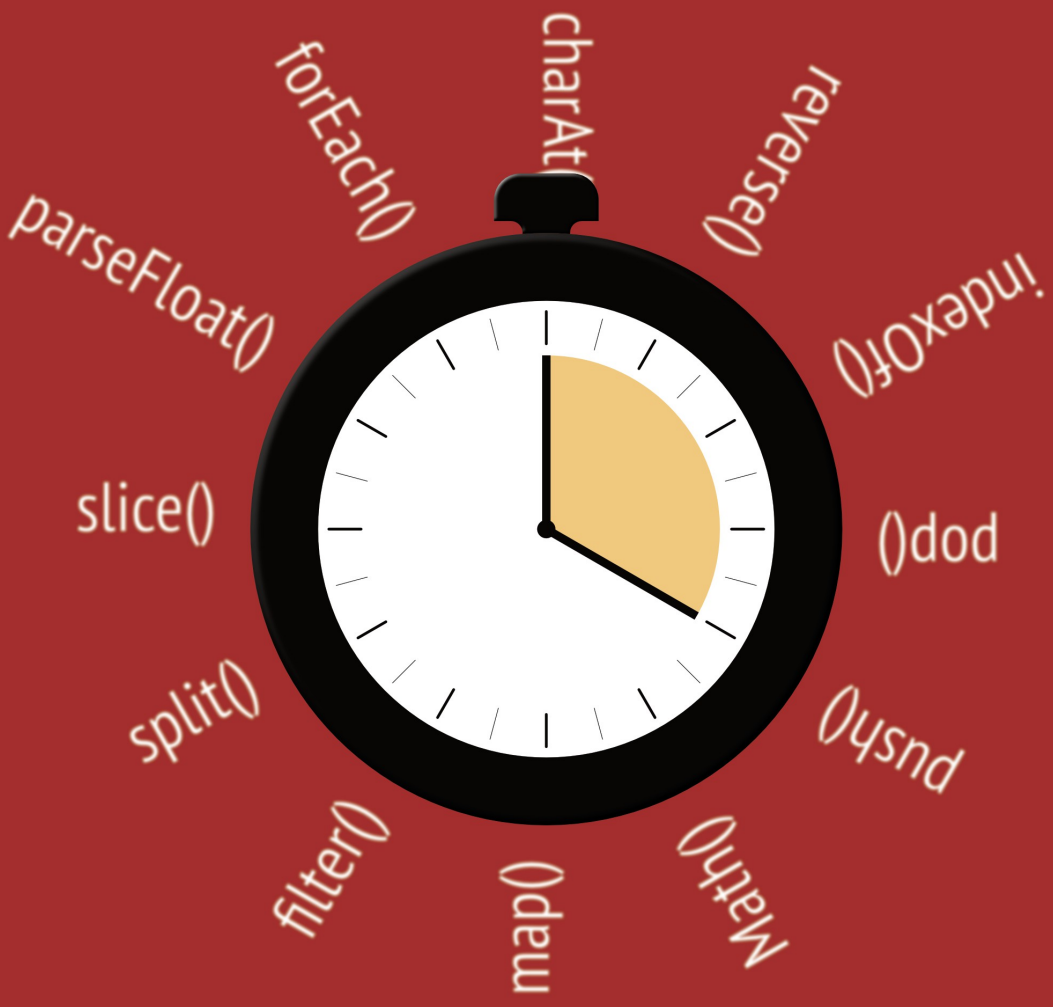


Table of Contents

Instructions

One trait of a good developer is a person that can **write code**, not **copy and paste it only**. Or at least doing the former much more than the latter.

I was a copy and paste developer for a long time, relying heavily on Stack Overflow code that I would snatch and then alter to meet the conditions of my own purposes.

Now I'm not saying you can't do this, but you should reach a point where you can produce a lot of code from scratch, talking through it out loud and actually "writing code."

So in an effort to get better at typing code without looking everything up, AND in an effort to become better acquainted with the syntax and methods of JavaScript, I created this blueprint.

20 minute methods is a commitment to schedule out 12 days in order to devote 20 minutes to learning individual JavaScript methods one bite at a time; methods like split(), map(), slice(), join(), indexOf(), toString(), reduce(), etc.

You can complete the lessons in sequence, i.e. 12 days, or you can plan to do them every other day, perhaps 3 days a week and complete it all in 4 weeks. It's up to you.

Here's what you should do to get the maximum benefit of this book:

- Do the exercises in your browser console, preferably Chrome. . It's a wonderful place that you should become acquainted with.
- Devote 20 minutes of undivided attention to each lesson. Get the feel of typing that method, that array, that syntax from scratch. Build that muscle memory. Get the feel of what the parameters do. Try out your own combinations and tests until 20 minutes is up.
- Continue on your predetermined schedule until the book is done.
- Print out a quality copy of the cheatsheet and hang it up by your desk. After going through the methods once, the cheatsheet will make complete sense and will be a valuable lookup as you code in JavaScript.
- This is also offered as a free e-mail course. Sign up at: https://travis.media/20mm

Have fun with it! Let me know how it's going at contact@travis.media.

Regards,

Travis Rodgers
https://travis.media
Virginia

Day 1: split()

Hello, and welcome to the beginning of 20 Minute Methods - JavaScript.

I'm glad that you have decided to join me in getting more acquainted with JavaScript by looking at its methods, one bite at a time.

To kick things off, we are going to look at the JavaScript String split() method.

Note the word String.

The JavaScript split() method splits a string into an array of substrings

That's it.

It's now up to you to tell it where and how to split.

The split() method takes two arguments, both optional, and they are (separator, limit):

1. **Separator** - Determines the character to use for splitting the string. If left blank, the entire string will be returned intact, but as an array.

2. **Limit** - An integer specifying the number of splits. Items after this limit will not be included.

As I will make a habit of doing, instead of me explaining examples I'll give you a few to do to demonstrate.

Let's Go

Open up a browser and the console. Then type a variable such as:

`var today = "Today was a very productive day."`

Hit Enter.

Next enter: `today.split(" ", 2).`

Remember today.split will have two arguments, (separator, limit).

So the separator is a space, and the limit is 2 meaning it will return two words and discard the rest giving you:

["Today", "was"]

Super. We have an array of the first two substrings.

Now It's Your Turn

Now here are some exercises to try in the console.

Don't copy and paste this, but type it out. Get your muscle memory and your brain used to typing and using this split() method.

So try these in the console (and you will only need to type in your variable once):

Examples

`var string = "This is going to be fun looking into the string method";`

First, see what happens with `string.split();`

Now note the very different results of `string.split("");` versus `string.split(" ");` <--just a space difference. Wow.

More examples:

`string.split(",");` <--hmm there are no commas in our string, so nothing.

`string.split("g");` <-- Does this remove all the g's or just the first one? What is put in place of the g's? Why is this? (Because it creates the next index of the array).

What about this: `string.split("g", 3);` What happens after the 3rd g?

`string.split(" ", 4);` <--how nice. I can see many practical uses of that.

Here's a challenge for you:

`var tricky = "goshgoshgoshgoshgoshgoshgoshgo";`

See if you can make this say *["go", "go", "go", "go", "go", "go", "go", "go"]*

Good.

Now keep typing in stuff and trying out different combinations until you nail down this method.

Great job!

Day 2: slice()

Now for our second 20 Minute Method and it's called:

slice()

What is the JavaScript Array slice() Method?

First, note the word Array.

The slice() method returns the selected elements in an array, as a new array object. It "slices" out what you tell it to.

The slice() method has two parameters (start, end), both optional.

1. **Start** - An integer. This determines where to start the selection. The first item in the array has an index of 0. This is inclusive.

2. **End** - An integer. This determines where to end the selection. This is not included in the new array object. This is not inclusive (the new array object does not include the end argument).

As I will make a habit of doing, instead of me explaining examples, I'll give you a few to do to demonstrate.

Let's Go

Open up a browser and the console. Then type a variable for an array such as:

`var array = ["birds", "trees", "fish", "land", "sea", "boats"];`

Hit Enter.

Next enter: `array.slice();`

What happens?

Well, nothing as we didn't specify any slice.

Remember `array.slice()` will have two arguments, (start, end).

Now enter: `array.slice(1, 3);`

What do you get? Is "land" included? Why not?

Now It's Your Turn

Let's try a new array:

`var array = ["fire", "water", "ice", "steam", "sand", "grass"];`

Now try: *array.slice(0, 4);* What element is in index 0?

Let's try without a number:

array.slice(, 3); <-- does this work?

array.slice(3); Is this 3 the start or end number in the results?

Now take the time to try out your own combinations.

Here's a tricky one:

What are two ways of getting the last element of the array only (which is "grass")?

Try to figure it out first and then look below

Here are the two ways:

array.slice(5); <---gives us the 6th and final element

array.slice(-1); <--- -1 give us the last array as it starts from the right.

So how do we get the last two with a negative number?

You bet:

array.slice(-2);

And the final challenge:

How do we get "water", "ice", and "steam" with two negative numbers as the arguments start and end?

Try it out. Think it through and then check your answer below.

Answer:

array.slice(-5, -2); <--Remember that the second argument, the end, is not inclusive so it is not going to include "sand".

Hey, though this was an easy one, I hope it was helpful. You now have two methods under your belt that you can TYPE and USE from memory!!

Day 3: slice(), substring(), substr()

And now for the third lesson we'll look at:

slice(), substring(), and substr()

Wait!? We did slice() already and three is too much to learn in one day right?!

Nope.

Let me explain:

slice()

The slice() method that we learned in our last lesson was an Array method. It also can be used as a String method.

The difference is that **instead of slicing the array indexes, you are slicing the string characters**. Remember that it has two parameters (start, end). These are the numeric positions starting with 0.

For example:

var string = "Now, this is a string to be slicing";

string.slice(3, 8); <--This would give us a new string of ", thi"

Note that the whitespace and the comma are included.

So you know this method already. Just instead of slicing array indexes, you are slicing characters of a string.

substring()

This one is easy. Just remember this:

substring() is the exact same as slice(), with the only difference being that it can't accept negative values.

Enough said. You got this!

Now here is our new one for today.

substr()

Today we are going to look at another easy one:

substr()

What is the substr() method?

The substr() extracts parts of a string beginning at the character of the specified position, and returns the specified number of characters.

It has two parameters (start, length)

1. **start** - this is required. This is the starting position. Remember that the first character is 0.

2. **length** - optional. The number of characters to extract after the starting position. If empty, then it displays all of the characters after the starting position.

Let's Go

Open up a browser and the console. Then type a variable such as:

var string = "This is the end of week one";

Hit Enter.

Next enter: *string.substr(0, 1);*

It starts at 0, which is the letter T, and the length is 1, so it extracts 1 letter giving us "T"

Now try *string.substr(8);*

It starts at the 8th position, which is "t" (remember to count the whitespace), and since the length is left off it counts all of the remaining characters.

Now It's Your Turn

Let's try a new variable:

var string = "Now it's my turn to try it out";

Exercises:

(The answers will be at the bottom if you have trouble).

1. Try to produce only a new string of "out"

2. Try to produce only the last "t" of the entire string using a negative number.

Remember that the second parameter get's the length. So you start with the first and can tell it how many more characters to extract using the length parameter.

3. So in the above string, start with the letter "m" of my and extract all characters up until the "r" in try.

4. Now start at 0 and extract all of the characters up until the "s" in the word, it's (this means stopping on, and including, the apostrophe).

Great job, I think you got it.

Now you see why we did three methods today because you basically knew them already.

Answers

string.substr(27, 3); <--- I bet you may have tried accidentally (27, 29). This is wrong but it gives you the right answer. Why? Because "out" is the last word in the string, so since the length is greater than the string, it will give you all of it.

string.substr(-1);

string.substr(9, 13); <--- Remember that the letter "m" in "my" is going to be the first count of the length. You set the starting point and then you begin counting at the starting point.

string.substr(0, 7); <--- 0 is the starting character ("N"), and the length is "N"(1), "o"(2), "w"(3), " "(4), "i"(5), "t"(6), ""(7).

Day 4: reverse() and join()

Let's kick off lesson 4 mixing a String and two Array methods, something new....but related to what we learned in the past three lessons.

We need to stay fresh!

So in this lesson we will use the String slice() method (which we learned last week), and combine it with two new Array methods:

`reverse()` and `join();`

reverse()

This is a JavaScript Array Method.

Note the word Array.

The reverse() method reverses the order of an array.

It has no parameters, it just simply reverses.

Let's Go

Open up a browser and the console. Then type a variable for an array such as:

`var array = ["r", "e", "t", "u", "p", "m", "o", "c"];`

Hit Enter.

Next enter: *`array.reverse();`*

Simple. It reverses the elements of the array giving you:

["c", "o", "m", "p", "u", "t", "e", "r"];

That's it!!

Now let's look at the next one:

join()

The join() method joins the elements of an array into a string, and returns a string.

It has one parameter (separator).

This parameter is where you specify what you want the elements to be separated by. The default is the comma (,).

Let's Go

Let's use the same variable from our above example.

var array = ["r", "e", "t", "u", "p", "m", "o", "c"];

Hit Enter.

Next enter: *array.join();*

What do you get? What is the default separator?

Now enter: *array.join("-");* What do you get?

Want spaces between the words? Try *array.join(" ");*

Now experiment. Try a pipe |. Try letters, symbols, etc.

Here's a challenge. Try to get:

"r] [e] [t] [u] [p] [m] [o] [c" <--- be sure there are spaces!

Your Challenge For Today

Let's try to combine a few things here for our challenge.

Remember slice()? It takes two arguments (separator, limit).

Here is your challenge:

Take this string:

"tnellecxe"

and reverse it into a new string "excellent"

Hint: You can do this by using the String slice() method, the Array reverse() method, and the Array join() method.

Give it a shot. Use any variables you wish.

The answer will be below so check it out after you attempt or complete it. Also, there will be a bonus challenge below.

Answer:

var splitString = "tnellecxe"

//First we need to split() the characters of the string into array elements.

var elementArray = splitString.split(""); //This gives us ["t", "n", "e", "l", "l", "e", "c", "x", "e"]

//Now we need to reverse this array.

var reversedArray = elementArray.reverse(); //This gives us ["e", "x", "c", "e", "l", "l", "e", "n", "t"]

//Now lets join our reversed array back into a new string

var joinArray = reversedArray.join(""); //This gives us "excellent"

Bonus Challenge

Take this string:

"today work Great"

and reverse it into a new string saying "Great work today"

If you get stumped on this one, shoot me an email at contact@travis.media and I'll send you an answer.

Great work!

By now you know split(), slice(), substring(), substr(), reverse, and join().

That's some good progress!

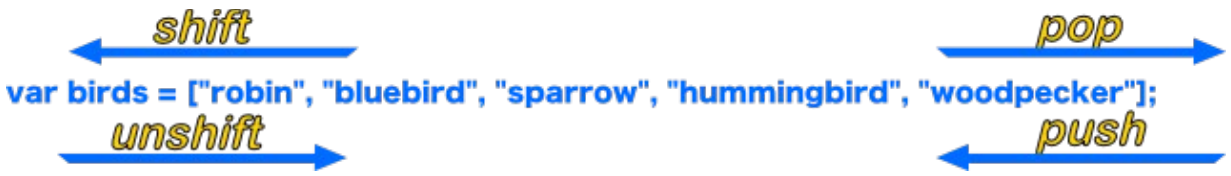
Day 5: shift(), unshift(), pop(), push()

I hope learning this bit-by-bit look at JavaScript methods has been helpful to you thus far.

Today we are going to jump over to arrays and cover a few fundamental Array methods that you will use a lot. Now you may think you already know these, but rest assured there are a couple of things you have to grasp before you can claim this. I will explain.

These methods are: pop(); push(); shift(); and unshift();

Here is a simple diagram explaining the gist of it:



Let's take a look at each one, starting with the two at the front of the array:

**Remember, the shifts manipulate the front of arrays. Try to remember this with this phrase, "Everyone likes to work first shift!" Cause that's true right?

shift() and unshift()

The shift() method removes the first array element and "shifts" all of the other elements to a lower index.

Simple right? Yes, but there are a few things to note.

Let me explain with an example:

Open up a browser and the console . Then type a variable for an array such as:

`var array = ["one", "two", "three", "four", "five", "six"];`

Hit Enter.

Now enter: `array.shift();`

What do you get?

It returns "one" which is the first element of the array. All the other elements move down. Your array (that same array not a new one) no longer includes "one."

Simple.

Now if you wanted to add it back OR add a different element to the front of the array, just use unshift():

`array.unshift("one");`

But wait.....why am I getting a 6 when I put that in. Because the **unshift() not only adds to the front of an array, but it returns the number of elements in an array**.

So just think how that could be used! If you entered array.unshift() with no argument, it simply returns the number of elements in an array. Neat! Write that down because you will be using it often in your coding career.

pop() and push()

Now let's look at the end of an array with pop() and push().

Think about holding the array in your hands and "popping the end off." You pop something off. Remember that and it will follow that both "p" methods concern the end of the array, pop and push.

But if you took the time to understand shift and unshift, then you will find these two easy.

The pop() method returns the array element that is "popped off" the end of the array.

For example:

Type in your console `var array = ["one", "two", "three", "four", "five", "six"];`

Hit Enter.

Now enter: `array.pop();`

It pops off the last element of the array, and returns that value which is "six".

Now if you check, you see that "six" is missing. Try it, enter `array` to see.

If you wanted to add it back OR add a different element to the front of the array, just use push():

`array.push("six");`

The push() method adds back the "six" and returns the array count which is 6. (Which again is a valuable way to check the number of elements in an array).

Check the array and see that its back on the end. Try it, enter `array` to see.

And that's it. Now go and try it out with new arrays until you feel comfortable. You will use these a lot, especially with loops.

Well hey, four methods in one day isn't bad at all!!!

Day 6: charAt(), indexOf(), lastIndexOf()

Now let's switch it back to Strings and do a little searching with the following three methods:

charAt(); indexOf(); lastIndexOf();

Let's get started.

charAt():

The charAt() method returns the character at a specified index of a string.

It has one parameter, (index)

So our parameter is an index position, and it returns the letter in that position. Just remember this: What position is such-and-such character at (char at)?

Example:

Open up your console and type in the following string:

var position = "I want to search for positions within this string.";

Hit Enter.

Now pick an index number. Let's say 15.

position.charAt(15);

We get "h". Great!

What do you get with *position.charAt(0);*

Yep.

Simple. Next.

indexOf();

Now where charAt() gives us the value of the position we pass as an argument, indexOf() gives us the position of a value.

In one instance we are getting the character at, and the other we are getting the index of.

indexOf() takes two arguments:

1. **The search value** - the string we are searching for

2. **The starting index** - the index from where we are to start the search. This is optional.

...so indexOf(searchValue, startingIndex);

Example:

Let's work with this variable:

var sleep = "I am going to go to sleep after this.";

Remember with indexOf() we are getting the index of a value in the string.

So *sleep.indexOf("am");* is going to give you 2 as it begins with an "a" and is in the second position (or index).

Now let's try the second argument and say we want to get the first occurrence of "to" after position 5. Try out *sleep.indexOf("to", 5);*

We get 11 because the first occurrence of "to" after index 5 comes at index position 11.

But wait, why the first occurrence of "to?" What if I wanted to get the last occurrence of "to" after 5. If you see in our sentence, we have the word "to" twice.

To get the last occurrence, we have to use:

lastIndexOf();

The lastIndexOf() method returns the position of the last occurrence of a specified value in a string.

Examples:

Try it out: *sleep.lastIndexOf("to");*

What do you get? Yes, the position of the last occurrence of "to" which is 17.

Now check this: Let's say we wanted to get the first occurrence of a "g" after index position 2.

Try it yourself and see what you get.

Wait, did you get -1???

Did you try *sleep.lastIndexOf("g", 2);?*

That was right but what went wrong?

Let me explain. The lastIndexOf(); method counts backwards but returns to us the index starting at the beginning. So in the above example of sleep.lastIndexOf("g", 2); it began in the second index (the "a" in am) and counted backwards. If you follow with your finger, you will see that there is no "g". Only a space and an I.

**When the starting point is not given for lastIndexOf();, then the default is the length of the string.

Now try to the get the last occurrence of the letter "g" starting from index position 28.

Great job.

Can You Comprehend This Chart Now?

Here is a chart by MDN that you should be able to follow along with now and your challenge for the day. If you struggle with this in any way, feel free to go back over the above until you are able to work it out:

```
'Blue Whale'.indexOf('Blue');     // returns  0
'Blue Whale'.indexOf('Blute');    // returns -1
'Blue Whale'.indexOf('Whale', 0); // returns  5
'Blue Whale'.indexOf('Whale', 5); // returns  5
'Blue Whale'.indexOf('Whale', 7); // returns -1
'Blue Whale'.indexOf('');         // returns  0
'Blue Whale'.indexOf('', 9);      // returns  9
'Blue Whale'.indexOf('', 10);     // returns 10
'Blue Whale'.indexOf('', 11);     // returns 10
```

Great!

Now breathe.....

That was quite a handful right?.

Side note.

If the string that you are looking for does not exist, then it returns -1.

This is a great way to check if something in a string exists.

And in addition, this same method, indexOf(); can be used in the same way with Arrays. This is a great method for searching out arrays to see if values exist by checking to see if it is equal to -1.

Okay, I think that is enough.

Day 7: String(), toString(), toFixed(), Number(), parseFloat(), parseInt()

Now before we get into some of the more complex stuff, we need to have a quick look at some Number methods and we will do that today and in the next lesson.

These are very important and ones you will use often.

Today we will be looking at number conversions:

String(); toString(); toFixed(); Number(); parseFloat(); parseInt();

What? Why so many?

Well, because they are very easy and you can handle it!!

Let's break it into two camps:

1. Numbers to strings
2. Strings to numbers

Numbers to Strings

There are times where you need to convert a number to a string:

toString() and String();

These will be used a lot and should be pretty clear.

toString() AND String() **both convert a number to a string.**

Open up a browser and the console. Then type a variable such as:

`var numberToString = 500;`

Hit Enter.

Now let's check what type of variable this is. You do that by typing: `typeof numberToString`

This will return "number" (and as a side note, you can also check this in a console.log situation by entering console.log(typeof numberToString);

Now let's convert this number to a string in a new variable by using our toString() method.

Type:

var newString = numberToString.toString();

And we get "string"

Great! If we type in newString, our number 500 is now surrounded by quotes.....a string.

toFixed();

toFixed() does the same as toString() but it **returns the string with a set number of decimals** that you would pass in as an argument.

The most widely used is the money format which is two decimal places.

Let's try.

Type in:

var money = 3;

Let's check what type this variable is by entering: *typeof money*. We get "number"

Now if we wanted to add two decimal places we enter: *money.toFixed(2);*

We now get "3.00" which is a string. Feel free to check with typeof.

When I enter (2), I am telling it to place two decimal spaces. If we want 4 decimal spaces we put a (4) as the argument. If we want 6, we put (6).

Challenge

Now here is a challenge for you. What if we have this variable:

var money = 3.22;

If we put *money.toFixed(4)* we will get 3.2200.

But what happens if we put *money.toFixed(0)*??? What do you think would happen if your variable was 3.51 and we put *money.toFixed(0)*?

Try it out and see.

Strings to Numbers

Now, what if we want our strings to be numbers? Let's look at the most common three:

Number();

This is by far the simplest and does the opposite of String(). Enter your desired string as the argument. So:

If we have the string "10" and we need to convert it to a number we simply put:

String("10");

Done! ------> 10

parseInt()

parseInt() **parses a string and returns a whole number**

This is a great way to round a string into a whole number (with an exception as noted below).

So for example:

parseInt("3.22"); will give you 3.

But parseInt("3.77") will also give you 3.

Why? Because it will round down as it converts. This is important to use at times in software applications. For example, if your game gave out trophies for each level you pass and you have passed 4 and 3/4 levels, that is still just 4 levels. Even at 4.99, it is not quite 5 levels and thus should not award you 5 trophies.

parseFloat();

parseFloat() **parses a string and returns a number.**

So you will use this one, not parseInt(), if you want to convert the string to a number AND keep your decimal places.

If you type in *parseFloat("3.45");* then it will return the number 3.45. No rounding.

Review

That's it for today! Let me recap.

Number to string:

toString(); - converts a number to a string
String(); - converts a number to a string
toFixed(); - converts a number to a string with a parameter to set number of decimal places

String to number:

Number(); - converts a string to a number
parseInt(); - parses a string and returns a number that is rounded down.
parseFloat(); - parses a string and returns a number that keeps its decimal places (if it has them).

Great job. We will look at a few more number methods in the next lesson.

Day 8: Math.round(), Math.ceil(), Math.floor(), Math.max(), Math.min()

Today we will finish up our numbers portion of the course by looking at Math.

What is this Math you speak about?

Math is a built-in JavaScript object that allows you to perform mathematical tasks on numbers.

Here are five simple Math methods that you will use often:

Math.round(); Math.ceil(); Math.floor(); Math.max(); Math.min();

In fact, you can probably tell what they do by just reading them.

Let's dig now into this brief lesson:

Math.round();

The Math.round() method returns the value of a number rounded to the nearest integer.

Open up a browser and the console (if you are viewing this in a browser, just open up the console under this email). Then type a variable such as:

var number = 12.8;

Hit Enter.

Now lets round:

Enter *Math.round(number);*

And yep, you get 13.

You can also just enter the number as the argument and get the same outcome like:

Math.round(12.8); <----13

Math.round(12.3); <----12

Now try these numbers out and get a feel for how it rounds to the nearest integer:

12456.33211
192.88111

Great!

Math.ceil(); and Math.floor();

Now for these two just remember this:

The ceiling is up and the floor is down.

Math.ceil() returns a value rounded up to its nearest integer.

Math.floor() returns a value rounded down to its nearest integer.

That's it!

Math.ceil(4.1); <---- 5

Math.floor(4.9); <---- 4

Simple enough.

Math.min() and Math.max()

The min() and max() methods do just what they suggest: **They return either the lowest number of the numbers passed to it (min) or the largest number of the numbers passed to it (max).**

Math.min(4, 2, 9); <---- will return 2

Math.max(4, 2, 9); <---- will return 9;

Let's try another:

var array = [3, 2, 5.4, 10, 3.2, 11, 20, 1, 0];

Now enter *Math.min(array);*

What did you get? 0?

Nope, you got NAN.

Why. Well, let's look at the parameters of this method.

Math.min([value1[, value2[, ...]]])

Look at those three dots. With ES2015, we can use this to pass an array. It is called a spread operator. This actually causes the array values to be expanded, or "spread", into the function's arguments.

Try it out:

Math.min(...array);

Math.max(...array);

Great job. And that's all for today. Go and rest your brain because our last four lessons are going to be quite a bit tougher.

Day 9: forEach()

Now its time to take things up a notch for our last four lessons.

Today we are going to look at the JavaScript forEach() array method and it's going to be a lot of fun.

Now, while it is similar, the forEach method is not a replacement for the 'for' loop. There are pros and cons to each, and time will teach you when to use one over the other.

That being said, the forEach loop is fun, it is less code, and it is much cleaner.

Let's take a look at it:

forEach();

The forEach method executes a provided function once for each array element.

Now what does that mean?

Well, let's look at a simple example and see.

Let's create an array that we can use.

Open up a browser and the console. Then type in the following array:

```
var numbers = ["5", "4", "3", "2", "1", "0"];
```

First, we start with our array, and add on our method like so:

```
numbers.forEach();
```

Second, we pass in a callback function:

```
numbers.forEach(function() {

})
```

Third, we pass in our element as the iterator, and this can be any name you like. Sometimes I just use 'element' and other times I try to use the singular form of the variable. In this case, we will do the latter and use 'number'. So:

```
numbers.forEach(function(number) {

})
```

Fourth, let's console.log our iterator and see what we get:

```
numbers.forEach(function(number) {
    console.log(number)
})
```

Great!

In this instance, it is much like a for loop but much cleaner and easier to write.

Your Turn

Let's use what we've learned to push these array elements into a new array.

So here is our number array again, enter it into your console:

```
var numbers = ["5", "4", "3", "2", "1", "0"];
```

And let's set it up this way:

```
var numbers = ["5", "4", "3", "2", "1", "0"];
var newNumbers;
```

Now I want you try to push the array elements of "numbers" into the empty "newNumbers" array using the forEach method.

Go!

Ok, let's walk through it again.

First, we start with our variable, and add on our method like so:

```
numbers.forEach();
```

Second, we pass in a callback function:

```
numbers.forEach(function() {

})
```

Third, we pass in our element as the iterator. So:

```
numbers.forEach(function(number) {

})
```

Fourth, for each element of the 'numbers' array, we will push it into the 'newNumbers' array:

```
numbers.forEach(function(number) {
    newNumbers.push(number)
})
```

Now check the contents of newNumbers by typing 'newNumbers' in your console and you will see that it now contains our six numbers.

Good job!

Advanced Method

As you may have realized, instead of using a callback function in the forEach method, you can simply call an already established function.

Let's say we have this function that multiplies by 3 and spits out the value to the console for us to see the results:

```
function sumFunction(x) {
    x *= 3;
    console.log(x);
}
```

We can actually call this function on each one of our array elements like such:

First enter: `var numbers = ["5", "4", "3", "2", "1", "0"];`

Then enter:

```
function sumFunction(x) {
    x *= 3;
    console.log(x);
}
```

Finally enter: `numbers.forEach(sumFunction);`

And there we used our forEach method to call a function on each element of our numbers array.

**You cannot break a forEach loop or throw an exception. If you wish to do this, use the for loop instead.

Now Choose

This is how you will use the forEach loop the majority of the time, much like a 'for' loop. However, in addition to the element argument we have been using, there are two more arguments. If you want to learn these, keep reading. If not, I think the forEach loop will serve you well and you can stop here. If in doubt.... keep reading!!

Lets Go!

So let's look back at the first two steps of our forEach method:

First, the base: `numbers.forEach();`

Next, the callback function:

```
numbers.forEach(function(){

});
```

Now, here are the three parameters:

1. **element** - we have looked at this. This is the current array element. In our example above it was 'number'; (required).

2. **index** - the array index of the current element (optional).

3. **arr** - the original array object, if needed (optional).

Example:

Let's look at a simple example. Here is our array:

First enter: `var numbers = ["5", "4", "3", "2", "1", "0"];`

Enter that in the console.

Now try this one out:

```
numbers.forEach(function(element, index, array){

   console.log('Element:'  + element + " Index: " + index + " Array: " + array);

});
```

Now what is all that?

Basically, for each element of our array we returned the Element, the IndexPosition, and the original array.

Example 2:

So let's say we were working on a project, and we needed to know the contents of an array and exactly where the index of 'orange' was. And let's say this is our array:

```
var fruits = ['kiwi', 'apple', 'banana', 'pear', 'grapes', 'lime', 'orange', 'lemon' ];
```

Here is a challenge for you. See if you can produce the following result. Then check the answer below:

Kiwi is in index 0
Apple is in index 1
Banana is in index 2
Pear is in index 3
Grapes is in index 4
Lime is in index 5
Orange is in index 6
Lemon is in index 7

And go!

Answer

So let's take this in steps:

Again, here is our array: var fruits = ['kiwi', 'apple', 'banana', 'pear', 'grapes', 'lime', 'orange', 'lemon'];

First, we type out our method. (Remember, we are typing everything for muscle memory purposes).

```
fruits.forEach();
```

Second, we add our callback function:

```
fruits.forEach(function() {

});
```

Third, we will enter our needed parameters:

```
fruits.forEach(function(element, index) {

});
```

Fourth, we will console.log our answer to see each element applied:

```
fruits.forEach(function(element, index) {
    console.log(element + ' is in index ' + index);
});
```

And there you have it. Great job!!!

BONUS

Want another challenge? Try to produce this result (the first word all uppercase), and see the answer below when you are finished your attempt. You know how to do this. You learned it a week or two ago.

KIWI is in index 0
APPLE is in index 1
BANANA is in index 2
PEAR is in index 3
GRAPES is in index 4
LIME is in index 5
ORANGE is in index 6
LEMON is in index 7

Answer

Yep, you got it ---> by using .the toUpperCase() method!

```
fruits.forEach(function(element, index) {
    console.log(element.toUpperCase() + ' is in index ' + index);
});
```

Okay, okay, okay, I know. That is enough for today.

I hope that has been fun for you. I love this method as well as the next three as they not only simplify things if you come to understand them, but they open up another world of possibilities in JavaScript.

Day 10: filter()

Up to this point, you have learned 26 methods!!!

Great work!

Today we will be looking at a new method called filter().

filter()

The filter() method creates a new array from the original array elements that have passed a certain condition, leaving out the ones that fail.

Simply put, it filters through an array, choosing the ones that pass the test (which is provided by a function).

You can see the immediate practical use of this one.

Let's take a look at it. And the filter() method takes the same parameters as the forEach() method, so we will start our lesson with the most common parameter:

element - the value of the current element;

Open up a browser and the console. Then type an array such as:

```
var age = ["35", "14", "52", "21", "11", "80", "18", "17"];
```

Just like we did with forEach(), we will break this one down step by step.

First, we will create a new variable that will check this array for ages 18 and over. We will enter our array name (age) and add on the method (filter);

```
var adult = age.filter();
```

Second, we pass in a callback function:

```
var adult = age.filter(function(){

})
```

Third, we pass in our array element (which will be the value of the current element of the array as it filters through each):

```
var adult = age.filter(function(element){

})
```

Fourth, let's add our condition:

```
var adult = age.filter(function(element){
    return element >= 18;
})
```

Now, let's check our adult variable with console.log (or just type adult in the console):

```
console.log(adult);
```

And our array has been filtered for the ages 18 and up!

Great job!

Another Example:

What if we wanted to get all the ages between 18 and 60?

Give it a shot, this time set the function separate from the method and call the function from the method. See the answer below when you are done.

Answer:

First, let's define the function (which sets the condition):

```
function checkAge(element){
    return element >= 18 && element <= 60;
}
```

Second, we filter through our array with our function as the argument:

```
var adult = age.filter(checkAge);
```

Now lets check our variable with console.log(adult).

Great!

Now that is a powerful method! And an easy one.

But there's more

Just like there were more parameters to the forEach() method, the same is true for the filter() method. AND, those parameters happen to be the same:

1. index - the array index of the current element (optional)
2. arr - the original array object (optional)

An Example

Let's say we want to get an array of index positions for all of the elements that are 18 and up.

So instead of returning all of the elements that are 18 or higher, we are going to filter this array and return the index positions.

So our array again:

```
var age = ["35", "14", "52", "21", "11", "80", "18", "17"];
```

First,we start with our variable, and add on our method like so:

```
age.filter();
```

Second, we pass in a callback function:

```
age.filter(function(){

})
```

Third, we we will pass our element parameter AND our index parameter as we will need the value of each:

```
age.filter(function(element, index){

})
```

Fourth, we will add our condition and console.log() the index (you can also return it, but for the sake of our tutorial we will just log it to the console):

```
age.filter(function(element, index){
    if (element >= 18) {
    console.log(index);
    }
})
```

Wonderful!!

Bonus

Now here's a challenge. Take this array:

```
var dayOfTheMonth = ["3", "15", "22", "2", "8", "30", "23"];
```

And filter through it returning all the days of the month that come after the 15th, and return it in this format:

22 has an index of 2
30 has an index of 5
23 has an index of 5

See the answer below!

Answer

Let's do this one with the function separate.

So here's what I got:

```
var dayOfTheMonth = ["3", "15", "22", "2", "8", "30", "23"];



function secondHalf(element, index){
    if (element > 15) {
        return element + " has an index of " + index;
    }
}

dayOfTheMonth.filter(secondHalf);
```

Awesome!!!

And that's the filter() method!

Don't you love populating out sentences like that or is it just me?

Day 11: map()

Welcome to the second to last lesson of 20 Minute Methods - JavaScript.

Today is the day we look at that method, that if you come to understand well, will become a powerful tool in your JavaScript toolbox. And it's:

map()

Now, what is the map() method?

The map() method creates a new array with the results of calling a function for every array element.

That sounds sort of like the forEach method right? Well, there is a subtle, yet important, difference.

The forEach() method does not actually return anything but 'undefined.' It mutates each array element and is done, discarding the return value. An example may be something like iterating over each array element and saving the mutated values to a database....and done.

On the other hand, the map() method iterates over an array, transforms each element of the array, and returns a new array with the transformed values!! This new array is the same size and does not discard any element like filter() method does.

So let's jump right in:

map()

Open up a browser and the console. Then type an array such as:

```
var numbers = ["0", "1", "2", "3", "4", "5"];
```

Now what parameters do you think the map() method uses? Take a guess.

You got it:

1. element - the current element being processed in the array.
2. index - the array index of the current element.
3. arr - the original array object

And let's try something easy like multiplying each element by 2.

First, let's enter our array name (numbers) and add on the method (map);

```
numbers.map();
```

Second, we pass in a callback function:

```
numbers.map(function(){

})
```

Third, we pass in our array element and as usual, this can be any name you like. You will often see people use "num".

```
numbers.map(function(element){

})
```

Fourth, let's add our condition (and remember that this returns a new array, so **be sure to return your condition):**

```
numbers.map(function(element){
    return element * 2;
})
```

Good job!

A few things to note to be sure that you understand map():

1.Our example returned a new array, thus the use of return. If we were to save our method in a new variable, this variable would be a completely new array with our doubled amounts.

2.The forEach() method, on the other hand, cannot be returned. This is why we had to console.log our answers because once the array elements are mutated, it's done. You can see immediately where the use of map() would be a much better tool if you had to do any sort of debugging down the road.

Another example:

Let's try another example but with our new array stored in a variable.

I will let you give it a shot. Here is the challenge:

Take this array:

```
var numbers = ["9", "16", "25", "400", "900"];
```

... and instead of creating a callback function, pass in the Math.sqrt object method as the parameter (getting the square root of each element) and create a new array called *squareRootArray*.

Hint: Use the JavaScript Math.sqrt object method. And remember, you want to pass your Math.sqrt as the parameter. See the answer below when you are done.

Answer:

First, let's take our array and add on the map() method:

```
numbers.map();
```

Second, let's pass the Math.sqrt object method as such, and make it a new variable called squareRootArray:

```
var squareRootArray = numbers.map(Math.sqrt);
```

Now hit enter...

Wait....undefined???

Well, that is because you stored it in a variable. Type `squareRootArray` and hit enter. There it is!

Note: You didn't alter the original array AND you have a brand new array with the transformed elements.

A third example:

Now let's get the index parameter involved.

Let's create a collection like such:

```
var people = [
   {firstname : "Jimmy", lastname: "Brown"},
   {firstname : "Cindy", lastname: "Malcolm"},
   {firstname : "Timothy", lastname: "Stumps"}
];
```

And let's create our function separate and pass in our current element and index. We'll call our function listNames.

```
function listNames(element, index) {
   var fullnames = [element.firstname, element.lastname];
   return fullnames;
}
```

So as we map our array, this function will return the first name and last name of our collection.

```
people.map(listNames);
```

Hmm, this works well, but lets dress it up a bit.

See if you can join (hint) them together in one line, in a new array called "names" with the index position listed before it like so:

```
"0 : Jimmy Brown", "1 : Cindy Malcolm", "2 : Timothy Stumps"
```

Give it a shot and then consult the answer below.

Answer

Okay, first let's define our function separately.

```
function listNames(element, index) {
    var fullnames = [element.firstname, element.lastname];
    return fullnames;
}
```

But we want to join them in one line so lets add on our join() method:

```
function listNames(element, index) {
    var fullnames = [element.firstname, element.lastname].join(" ");
    return fullnames;
}
```

And then lets add our index value nicely to the front:

```
function listNames(element, index) {
    var fullnames = [index + " : " + element.firstname, element.lastname].join(" ");
    return fullnames;
}
```

And finally, our map method with our function as the argument and assigned to our new variable:

```
var names = people.map(listNames)
```

Now check the variable by typing in names into the console.

Yes.....!

Great work!!

Important Recap

So listen closely. If you don't remember all the examples of this lesson, be sure to remember this:

The difference between map() and forEach() is that map actually returns something (a new array), while the forEach() method returns nothing (undefined).

This is key. Once you have that, just remember that map() iterates over an array, calling a function for each element, and returning the transformed elements into a new array of the same size.

And that's all there is to it.

Go and create some other arrays and get creative. Keep at it until you have a good understanding of the map() method.

See you on Friday for our final lesson and it's by far the neatest.

Day 12: reduce() and reduceRight()

Here it is, the last lesson. After today you will know 30 JavaScript methods!!!

So the last method we will learn is a complicated one up front, but once the light bulb goes off, you will find it easy to grasp. So take the time to slowly soak it in. The reward is great.

Today's method is the **reduce()** method. We will also look at the reduceRight() method.

reduce()

The reduce() method executes a function for each value of the array, from left to right, and reduces the array to a single value.

To understand this let's just walk through the format step by step:

First, let's build out the function and discuss what it does. Let's say we have an array called arrayExample. Let's attach our method to it.

```
arrayExample.reduce()
```

Second, let's enter our callback function:

```
arrayExample.reduce(function(){

})
```

Third, let's talk about the parameters as this is what trips everyone up:

The callback function has four parameters:

1. **accumulator** - accumulates all of the callbacks' returned values (required).
2. **element** - value of the current element (required)
3. **index** - index of the current element (optional
4. **arr** - the original array object (optional)

The good news is that you know #2, #3, and #4 already. They are the same as filter(), and forEach(), and map().

But what is this accumulator?

Well, think about this: If we are calling a function on each element, and these elements will be reduced into one single value, there needs to be a specified value where the accumulation happens, a value that totals up the changes into a single value. This is the accumulator!!

You will often see this called 'total' or 'acc'. Let's use acc in our examples.

So let's tally it up:

```
arrayExample.reduce(function(acc, element, index, arr){

})
```

Let's try out a simple example to start:

Example 1

Let's take an array, and reduce the elements into one final sum.

Open up a browser and the console. Then type an array such as:

```
var numbers = [0, 1, 2, 3, 4, 5, 6];
```

First, add the method to our array:

```
numbers.reduce();
```

Second, add the callback function that will be called on each element:

```
numbers.reduce(function(){

});
```

Third, we will enter our accumulator (required) and our current element (required) as arguments:

```
numbers.reduce(function(acc, element){

});
```

Finally, we will enter a condition into our function that adds the elements.

```
numbers.reduce(function(acc, element){
    return acc + element;
});
```

And we get 21!

Great!

Now here is what it is doing, broken down:

callback	accumulator	element	index	arr	return value
first call	0	0	0	[0, 1, 2, 3, 4, 5, 6]	0
second call	0	1	1	[0, 1, 2, 3, 4, 5, 6]	1
third call	1	2	2	[0, 1, 2, 3, 4, 5, 6]	3
fourth call	3	3	3	[0, 1, 2, 3, 4, 5, 6]	6
fifth call	6	4	4	[0, 1, 2, 3, 4, 5, 6]	10
sixth call	10	5	5	[0, 1, 2, 3, 4, 5, 6]	15
seventh call	15	6	6	[0, 1, 2, 3, 4, 5, 6]	21

Now there is ONE MORE parameter and it's called:

initialValue;

So we then have:

```
arrayExample.reduce(function(acc, element, index, arr){

}, initialValue)
```

And simply put, the initialValue is the value by which the accumulator will start on.

Let's try an example using the initialValue:

Example 2

So based on our example above, here is our array again:

```
var numbers = [0, 1, 2, 3, 4, 5, 6];
```

...and here is our reduce function with the initialValue of 8:

```
numbers.reduce(function(acc, element){
    return acc + element;
}, 8);
```

Now what this does, is it starts the accumulator at 8, giving us a return value of 29. Here is the breakdown showing the difference. See how the accumulator begins with 8?:

callback	accumulator	element	index	arr	return value
first call	8	0	0	[0, 1, 2, 3, 4, 5, 6]	8
second call	8	1	1	[0, 1, 2, 3, 4, 5, 6]	9
third call	9	2	2	[0, 1, 2, 3, 4, 5, 6]	11
fourth call	11	3	3	[0, 1, 2, 3, 4, 5, 6]	14
fifth call	14	4	4	[0, 1, 2, 3, 4, 5, 6]	18
sixth call	18	5	5	[0, 1, 2, 3, 4, 5, 6]	23
seventh call	23	6	6	[0, 1, 2, 3, 4, 5, 6]	29

So if there is no initialValue provided, the accumulator starts as the value of the first array element.

If there is an initialValue, this will be the initial value of the accumulator from the start.

Okay, not too bad.

Example 3

Let's look at a final example to assure that we understand the basics of reduce().

Let's look at some data:

```
var population = [
  {
    state: 'California',
    pop: 39780000,
  },
  {
    state: 'Virginia',
    pop: 8412000,
  },
  {
    state: 'Florida',
    pop: 20610000,
  },
  {
    state: 'Maine',
    pop: 1331000,
  }
```

```
]
```

Let's use the reduce() method to get the total population of our four states.

Why don't you give it a shot? Add up the populations (using element.pop) using reduce(). See the answer below.

First, we'll attach the reduce() method:

```
population.reduce()
```

Second, our callback function:

```
population.reduce(function(){

})
```

Third, our arguments:

```
population.reduce(function(acc, element){

})
```

Fourth, our conditions:

```
population.reduce(function(acc, element){
    return element.pop + acc;
})
```

Now what did you get?

Did you get 1331000206100008412000?

If you did, you are wrong, BUT this is a great thing because the lesson will much more beneficial than the mistake.

You see, if you did not set an initial value, then you actually started off with the number 39780000 initially!!!

Instead, we need to start off with an initialValue of 0.

So try it with the 0.

```
population.reduce(function(acc, element){
    return element.pop + acc;
}, 0)
```

..and you see that we now get the correct answer of 70133000.

Great work in tackling an initially difficult, but extremely powerful, method.

One last thing:

Now that you know the reduce() method, you can easily add to your arsenal reduceRight():

reduceRight();

reduceRight() is the exact same as reduce() but it reduces the array elements from right to left instead of left to right.

Done.

And that's all folks!!

I hope you have benefitted greatly from this bit by bit study of JavaScript methods.

If you have, please shoot me an email (contact@travis.media or hit reply to this email) and let me know of any further suggestions or feedback you would like to share. It would be highly welcomed and appreciated.

TRAVIS MEDIA
https//travis.media
contact@travis.media