Introduction to the JavaScript Array map() method

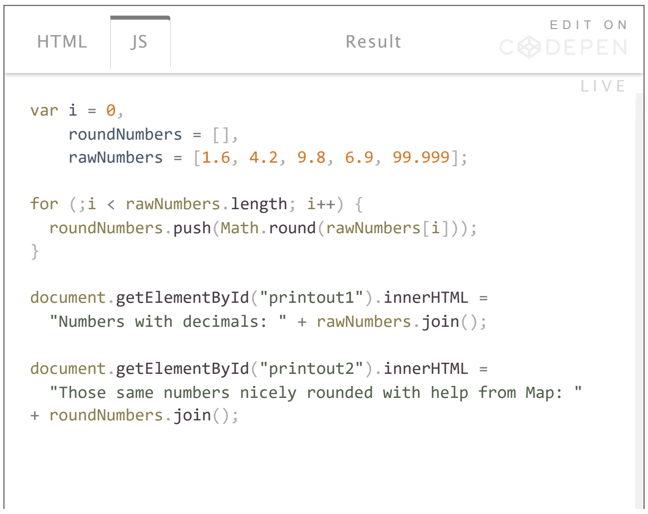
JavaScript's array map() method allows you to "do something" with each element of an array. This has always been possible, but the map() method's syntax is simple and elegant.

When you have an array, you'll eventually want to "do something" with it. You may, for example, want to do something with the entire array, such as showing all of its elements in the UI. But what about when you want to "do something" with each element in that array? This poses a bit more of a challenge.

First, you'll need to iterate the array, which requires some kind of loop. Second, you'll need to keep track of your iteration process, which requires a counter variable (which requires you to control that variable's scope). This may seem like no big deal, but it's work and each time you want to solve the same problem, you're writing code that's virtually identical, but not entirely the same, so you start to copy and paste.

This kind of repetitive boilerplate code is tedious, it must be managed, and as repeated code, it becomes a red flag. So, since this kind of coding presents unnecessary problems, what we'll cover in this article, is how the JavaScript **Array map()** method solves these issues.

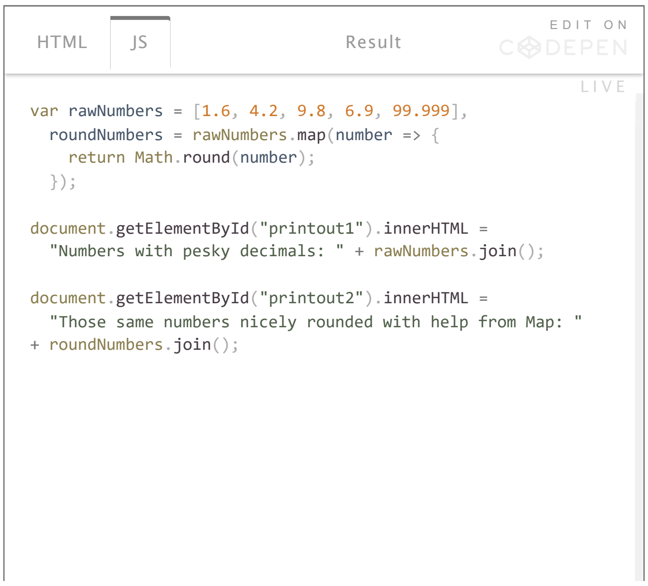
Using a for-loop – Example # 1



```
var i = 0,
    roundNumbers = [],
    rawNumbers = [1.6, 4.2, 9.8, 6.9, 99.999];

for (;i < rawNumbers.length; i++) {
  roundNumbers.push(Math.round(rawNumbers[i]));
}

document.getElementById("printout1").innerHTML =
  "Numbers with decimals: " + rawNumbers.join();

document.getElementById("printout2").innerHTML =
  "Those same numbers nicely rounded with help from Map: "
+ roundNumbers.join();
```

Our first pass at this solution employs a **for-loop**. As mentioned above, this introduces a few problems. We have the "**i**" variable, for instance, which in this example is a global. In a real-world situation, however, we would want to put this code in a function so that the "**i**" variable does not wind up in the global scope. Also, the code in our for-loop is tedious. We need to use the "**i**" variable to keep track of the currently iterated **rawNumbers** element. There's definitely a better way to go about this.

Using the Array map() method – Example # 2

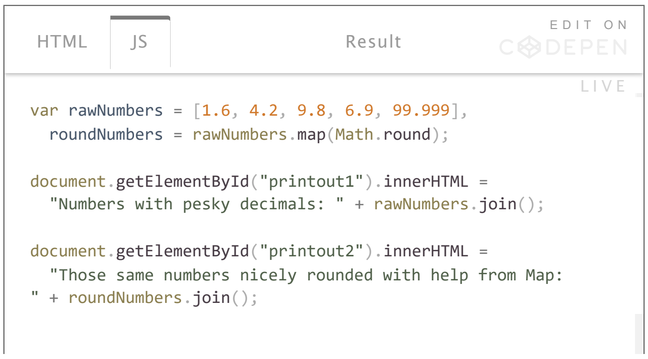


```
var rawNumbers = [1.6, 4.2, 9.8, 6.9, 99.999],
  roundNumbers = rawNumbers.map(number => {
    return Math.round(number);
  });

document.getElementById("printout1").innerHTML =
  "Numbers with pesky decimals: " + rawNumbers.join();

document.getElementById("printout2").innerHTML =
  "Those same numbers nicely rounded with help from Map: "
+ roundNumbers.join();
```

In Example # 2, we use the JavaScript **Array map()** method to solve the problems discussed. We've passed an anonymous function to that method, and this anonymous function takes the currently iterated number as its first argument. Inside of our function, we have some simple code that rounds the currently iterated number.

The biggest benefit to using the JavaScript **Array map()** method is that we no longer have the "**i**" variable or the for-loop. This is already a major improvement, especially the reduction of code, since it means that it's simpler and easier to read. These are not minor details, and if this is a task that occurs multiple times in your application, you'll soon find that the benefit gained by reducing repetitive code will quickly become significant.

Passing a function to the Array map() method – Example # 3



```
var rawNumbers = [1.6, 4.2, 9.8, 6.9, 99.999],
  roundNumbers = rawNumbers.map(Math.round);

document.getElementById("printout1").innerHTML =
  "Numbers with pesky decimals: " + rawNumbers.join();

document.getElementById("printout2").innerHTML =
  "Those same numbers nicely rounded with help from Map:
" + roundNumbers.join();
```

The approach in Example # 3 is similar, but there's one big difference: instead of passing an anonymous function to the **Array map()** method, we pass to the **Math.round** method, which makes our code even easier to read. This is largely because we leave the implementation details to **Math.round** and eliminate even more code: our anonymous callback function. There are two reasons that we can do this: **Math.round** takes a number to round as its first argument, just like our anonymous function, and it returns the rounded number, just like our anonymous function. Simplified code is better code and in this case, we have Math.round to thank for that.