Advanced Array Operations

In this lesson, we'll kick things up a notch with advanced array operations.
Let's begin!

WE'LL COVER THE FOLLOWING

Arrays are often used to iterate through their elements and carry out some kind of operation on each item.

Earlier in this course, you saw JavaScript examples where a `for` loop was utilized to traverse all elements in a collection. Although it was not explicitly mentioned, in most cases an `Array` instance was used to represent a collection.

The `Array` type provides a number of operations to help iterative element

processing. The methods representing these operations accept two arguments:

> 1. a function processing a single element (first argument), and
> 2. an optional scope object (second argument) in which to run the function.

In the processing function, this second argument can be accessed through `this`.

The processing function must have the following form:

```
function processor(element, index, array)
```

`element` is the array element to be processed, `index` is this element's index, and `array` is the array instance being traversed.

The table below summarizes the array iteration methods:

Array iteration methods

Method	Description
`every()`	This method tests whether all elements in the array pass the test implemented by the provided function. The provided function should return a Boolean value indicating the result of the test.
`filter()`	This method creates a new array with all elements that pass the test implemented by the provided function. The provided function should return a Boolean value indicating the result of the test.
`forEach()`	Executes a provided function once per array element.
`map()`	Creates a new array with the results of calling the provided function on every element in this array.
`some()`	Tests whether some element in the array passes the test implemented by the provided function. The provided function should return a Boolean value indicating the result of the test.

The code in Listing 7-22 provides simple examples of using `every()` and `some()`:

Listing 7-22: using `every()` and `some()` as

Listing 7-22: using `every()` and `some()` as iterators

```
<!DOCTYPE html>
<html>
<head>
  <title>every(), some()</title>
  <script>
    var test = [12, 5, 3, 41, 23].every(isPositive);
    console.log(test); // true;
    test = [12, 0, 3, -41, 23].every(isPositive);
    console.log(test); // false;
    var test = [12, 5, 3, 41, 23].some(greaterThan10);
    console.log(test); // true;

    function isPositive(e, i, a) {
      return e > 0;
    }

    function greaterThan10(e, i, a) {
      return e > 10;
    }
  </script>
</head>
<body>
  Listing 7-22: View the console output
</body>
</html>
```

Using `forEach()`, `filter()`, and `map()` is just as easy as the other iterator methods, as shown in Listing 7-23:

Listing 7-23: using `forEach()`, `filter()`, and `map()` as iterators

```
<!DOCTYPE html>
<html>
<head>
  <title>foreach(), filter(), map()</title>
  <script>
    var nums = [1, 2, 3, 4, 5];
    nums.forEach(display);
    console.log(nums.filter(even).toString());
    console.log(nums.map(square).toString());

    function display(e, i, a) {
      console.log("a[" + i + "] = " + e);
    }

    function even(e, i, a) {
      return e % 2 == 0;
    }

    function square(e, i, a) {
      return e * e;
```

```
    }
  </script>
</head>

<body>
  Listing 7-23: View the console output
</body>
</html>
```

This code snippet produces this output:

console

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
2,4
1,4,9,16,25
```

Quite often you iterate through an array of elements in order to calculate an aggregate value, such as the sum of elements. The `Array` type provides two methods, `reduce()` and `reduceRight()`, that let you simplify these kinds of aggregate operations.

Both methods accept two arguments: a reduction function to invoke on all items (first argument), and an optional initial value (second argument) upon which the reduction is based.

The reduction function must have this form:

```
function reductor(prev, curr, index, array)
```

Evidently, `index` is this element's index, and `array` is the array instance being traversed.

The `prev` argument is the value previously returned in the last invocation of the reduction function, while `curr` is the current element being processed in the array.

Both methods work similarly, but while `reduce()` processes array elements from the first index to the last, `reduceRight()` starts from the end of the array and advances backward the first element.

Listing 7-24 demonstrates using these methods.

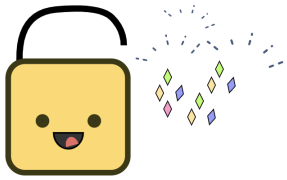
Listing 7-24: using the `reduce()` and `reduceRight()` methods

```
<!DOCTYPE html>
<html>
<head>
  <title>reduce(), reduceRight()</title>
  <script>
    var nums = [1, 2, 3, 4, 5];
    console.log(nums.reduce(sum));       // 15
    console.log(nums.reduce(sum, 20));   // 35
    console.log(nums.reduce(calc));      // 9
    console.log(nums.reduceRight(calc)); // 13

    function sum(prev, curr, i, a) {
      return prev + curr;
    }

    function calc(prev, curr, i, a) {
      return prev + (curr - i) * 2;
    }
  </script>
</head>
<body>
  Listing 7-24: View the console output
</body>
</html>
```

Achievement unlocked! 🎉



Congratulations! You've learned about some very useful advanced JS operations on arrays.

All around amazing work! Give yourself a round of applause! :)

By now, you learned a lot about the fundamental types of JavaScript. It is time to look after how you can create expressions.

In the *next lesson*, we'll cover unary operators.