array callback methods

map(), filter()

Array **callback methods** are methods called on arrays that take a function as their argument. This is in contrast to array methods such as **push()** and **pop()**, which do *not* have functions for arguments. We have actually already been working with array callbacks. The **sort(function(a, b))** for sorting arrays of objects by key.

array.map()

- the **map()** method is called on an array: **array.map()**
- the **map()** method call is set equal to a variable: **const newArr = array.map()**
- the **map()** method takes a callback function: **array.map(function() {})**
- the callback takes the current array item (default value: **e**) as *its* argument
- the callback performs some operation on **e** on each item in the array, one by one (as with a loop)
- the **map()** method returns an array of transformed items

1. Open **01.02-Map-and-Filter.html**, preview it in the browser and check the console. The output is the product of the the map and filter methods.

2. Switch from using **FINAL.js** to **START.js**.

3. Open **07.01-Map-and-Filter-START.js**, which is where we will recreate the **FINAL.js** file from scratch.

4. Declare an array so that we can try out the **map()** method:

```
const fruits = ['apple', 'blueberry', 'cherry'];
```

comparing map() to a for-loop

The **map()** method iterates over an array like a for loop but without the actual loop. So before we run the **map()** method on our fruits array, let's review iterating an array with a for-loop. The code in the loop will make fruit pies and push them into a **pies** array.

5. Declare a new empty array called **pies**.

```

```

```
const fruits = ['apple', 'blueberry', 'cherry'];
const pies = [];
```

6. Set up a for-loop that runs once for each item in the array:

```
const pies = [];
for(let i = 0; i < fruits.length; i++) {
}
```

7. Each time the loop runs, declare a variable **pie** and set the value to the current fruit, as **fruits[i]** followed by the word "pie":

```
for(let i = 0; i < fruits.length; i++) {
  let pie = fruits[i] + ' pie';
}
```

8. Also each time through the loop, push the new pie into the array:

```
const pies = [];
for(let i = 0; i < fruits.length; i++) {
  let pie = fruits[i] + ' pie';
  pies.push(pie);
}
```

9. Below the loop, log **pies** to the console:

```
  pies.push(pie);
}
console.log(pies);
```

Now, for the **map()** version of our "pie maker"; it gets the job done without a loop:

10. Call the **map()** method on the **fruits** array, setting it equal to a new array, **piesArr**, which will receive the return value of the **map()** method:

```
const piesArr = fruits.map();
```

11. Pass a callback function to the **map()** method. The callback takes an argument of its own, **e**, which represents the current fruits array item:

```
const piesArr = fruits.map(function(e) {
});
```

12. Inside the curly braces, concatenate and **return** the pie, all in one line. The **return** pushes the result into the new array:

```
const piesArr = fruits.map(function(e) {
  return e + ' pie';
});
```

13. Log the new array to verify that it worked:

```
console.log(piesArr);
// ['apple pie', 'blueberry pie', 'cherry pie']
```

calling the argument something besides e

The callback function argument is **e** by default, but we could call it anything.

14. Run **map()** again, saving the return value to a new variable and with **fru**, short for fruit, as the callback argument, instead of **e**. It works exactly the same as before:

```
const piesAgain = fruits.map(function(fru) {
  return fruit + ' pie';
});

console.log(piesAgain);
// ['apple pie', 'blueberry pie', 'cherry pie']
```

two arguments: e and i

In addition to the current item, default value **e**, the current array **index** is available, with a default value of **i**.

15. Run **map()** again, saving the return value to a new array again, but this time with the fruit pies numbered. To number them, use the index plus 1. The concatenation is getting longer, so switch to **string interpolation**:

```
const numberedPies = fruits.map(function(e, i) {
  return `${i+1}. ${e} pie`;
});

console.log(numberedPies);
// ['1. apple pie', '2. blueberry pie', '3. cherry pie']
```

array.filter()

- the **filter()** method works like **map()**, except that **filter()** returns an array of fewer items than in the original array
- the items in the new array may not be transformed in any way: it's the same items as in the original array--just fewer of them.
- the **filter()** method uses boolean comparison logic to evaluate the items. Only evaluations that **return true** will result in the time being saved to the new aray.

Let's save to a new array only five-letter fruits. First, we'll do it with a loop:

16. Given a new array called fruitsArr that includes several five-letter items, declare a new, empty array to hold the five-letter fruits:

```
  const fruitsArr = ['apple', 'banana', 'cherry', 'grape', 'lemon',
'lime', 'mango', 'papaya', 'peach', 'orange'];

  const fiveLetterFruits = [];
```

17. Loop the array:

```
  for(let i = 0; i < fruitsArr.length; i++) {
  }
```

18. With each iteration, check if the length of the current string equals 5:

```
  for(let i = 0; i < fruitsArr.length; i++) {
    if(fruitsArr[i].length == 5) {
    }
  }
```

19. If the condition is true, push the fruit into the **fiveLetterFruits** array:

```
  for(let i = 0; i < fruitsArr.length; i++) {
    if(fruitsArr[i].length == 5) {
      fiveLetterFruits.push(fruitsArr[i]);
    }
  }
```

20. Log the **fiveLetterFruits** array:

```
  console.log(fiveLetterFruits);
  // ['apple', 'grape', 'lemon', 'mango', 'peach'];
```

Next, let's make same array using **filter()**, instead of a loop:

21. Call **filter()** on **fruitsArr**, setting the expression equal to a variable which will "catch" the returned items:

```
const fiveChars = fruitsArr.filter(function(fru) {
  if(fru.length == 5) {
    return fru;
});
```

22. Log the result:

```
console.log(fiveChars);
// ['apple', 'grape', 'lemon', 'mango', 'peach'];
```

23. It worked, but we can simplify things. The filter method returns true comparisons, so the if part is not necessary. Just return the comparison itself:

```
const fivers = fruits.filter(function(fru) {
  return fru.length == 5;
});

console.log(fivers);
```

chaining methods: filter() into map()

Suppose we want pies of only five-letter fruits. This requires **filter()** to save the five-letter fruits, and then **map()** to add "pie" to the string. We can run the two methods one right after another, with the second method called diretly on the first. This technique is called **chaining**.

24. Declare an array called **char5Pies** and set it equal to the filter part:

```
const char5Pies = fruitsArr.filter(function(e) {
  return e.length == 5;
})
```

25. Chain the **map()** method onto the end of the **filter()** method:

```
const char5Pies = fruitsArr.filter(function(e) {
  return e.length == 5;
}).map(function(e) {
```

```
    return e + ' pie';
  });
```

26. Log the resulting **char5Pies** array. It should be piess of just five-letter fruits:

```
  console.log(char5Pies)
  // ['apple pie', 'grape pie', 'lemon pie',
  'mangp pie', 'peach pie']
```

three arguments: e, i and a

In addition to the current item **e** and the current index **i**, there is a third argument available, and that is **a**, the array itself.

In this next example, we will use **map()** with all three arguments to make strings of consecutive items joined by a hyphen ('apple-banana', etc.). These will be saved to a new **smoothies** array.

27. Call **map()** on the array, setting it equal to a new array, **smoothies**:

```
  const smoothies = fruitsArr.map();
```

28. Pass in the callback function with all three arguments: **(e, i, a)**:

```
  const smoothies = fruitsArr.map(function(e, i, a) {
  });
```

We need both **a** (array) and **i** (current index), so that we can get not only the current item, but the *next* item, too:

29. Inside the map function, concatenate and return the hyphenated, consecutive words combos:

```
  const smoothies = fruitsArr.map(function(e, i, a) {
    return e + '-' + a[i+1];
  });
```

30. Console log the result:

```
  console.log(smoothies);
  // ['apple-banana', 'banana-kiwi', 'kiwi-mango', 'mango-orange',
'orange-papaya', 'papaya-peach', 'peach-undefined']
```

31. To get rid of the 'peach-undefined', just pop it off the array:

```
  smoothies.pop();
  console.log(smoothies);
  // ['apple-banana', 'banana-kiwi', 'kiwi-mango', 'mango-orange',
'orange-papaya', 'papaya-peach']
```

filter() by object property

Given an array of objects, each a food menu item:

```
  const entrees = [
    { name: 'Chicken with Waffles', vegetarian: false, price: 18, cals:
1200 },
    { name: 'Tofuburger', vegetarian: true, price: 8, cals: 480 },
    { name: 'Porterhouse Steak', vegetarian: false, price: 29, cals: 1180
},
    { name: 'Quinoa Casserole', vegetarian: true, price: 14, cals: 560 },
    { name: 'Lobster', vegetarian: false, price: 28, cals: 750 },
    { name: 'Ribeye Steak', vegetarian: false, price: 32, cals: 1320 },
    { name: 'Banon Cheeseburger', vegetarian: false, price: 14, cals: 1400
},
    { name: 'Shrimp Scampi', vegetarian: false, price: 23, cals: 1060 },
    { name: 'Quinoa Burger Deluxe', vegetarian: true, price: 16, cals: 630
},
    { name: 'Chicken Salad Supreme', vegetarian: false, price: 13, cals:
710},
    { name: 'Salmon Steak', vegetarian: false, price: 22, cals: 680 },
    { name: 'Pork Chop', vegetarian: false, price: 18, cals: 800 },
  ];
```

We will use **filter()** to save to a new array all **non-vegetarian** entrees with a minimum **price** of 15.

31. Call **filter()** with its callback on arrays and set that equal to a new array:

```
  const nonVeg15 = entrees.filter(function(e) {
  });

  console.log(nonVeg15);
```

&& (AND) operator for filtering with two conditions

32. Use the **&&** operator to return non-vegetarian menu items that have a minimum price of $25. The two conditions go together in parentheses:

```
  const nonVeg$25Min = entrees.filter(function(e) {
    return (!e.vegetarian && e.price >= 25);
  });
```

```
console.log(nonVeg$25Min);
```

32. Use the **&&** operator to return non-vegetarian items under 1000 calories:

```
const nonVeg1000CalMax = entrees.filter(function(e) {
  return (!e.vegetarian && e.cals < 1000);
});
console.log(nonVeg1000CalMax);
```

END: Lesson 07.01 NEXT: Lesson 07.02