UNIT 07

LESSON 07.03

---

array.forEach()

array.find()

array.findIndex()

The **forEach** method is called on an array.

- Like **map** and **filter**, the **forEach** method passes the items of an array, one item at a time, to a callback function.
- Also like **map** and **filter**, the **forEach** method may be used instead of a loop to perform an operation on each item of an array.
- Unlike **map** and **filter**, **forEach** does not automatically return the result of that procedure.
- With no automatic return value, **forEach** is not subject to the strict rules of returning a new array with **map** ("must return the same number of items") or **filter** ("cannot transform the new items")
- The **forEach** method can be used as an alternative to map-into-filter chaining, since it can transform *and* filter items in an array, all in one go.

In this first example, we will square each number in an array of numbers and console.log the result.

1. Start with an array of numbers, and loop through it, logging each item to the console:

```
const nums = [3, 4, 5, 6, 7, 8, 9];
for(let i = 0; i < nums.length; i++) {
  console.log(i ** 2);
}
```

2. Now, refactor the code. Instead of using a loop, call **forEach()** on the array:

```
  nums.forEach();
```

3. Pass the **forEach** method an anonymous callback function, the argument of which represents the current item in the array. For this example, we will use **n**, representing the current **number** in the **nums** array:

```
  nums.forEach(function(n) {
  });
```

4. Log the square of the number to the console, and check the console.

```
    nums.forEach(function(n) {
      console.log(n ** 2); // 9, 16, 25, 36, 49, 64, 81
    });
```

5. It works, but better would be to refactor the callback function with arrow syntax:

```
    nums.forEach((n) => {
      console.log(n ** 2);
    });
```

6. With only one argument and one line of code inside it, the arrow syntax can be even more concise. Remove the argument parentheses and the function curly braces:

```
    nums.forEach(n => console.log(n ** 2));
```

forEach instead of map() and filter()
For this next example, let's call the forEach method on an array of strings and produce a new transformed array. This sounds like a job for map, so let's compare how the procedure is different with forEach.

7. Declare a new array of **things** strings, as well as a new empty array, **greens**:

```
  const things = ['Apple', 'Arrow', 'Bean', 'Coconut', 'Lantern', 'Light',
'Monster', 'Zone'];
  const greens = [];
```

8. Call **forEach** on the **things** array, and pass in the callback function, with **e** representing the current **element**:

```
  things.forEach((e) => {

  });
```

9. Add 'Green ' before the item, and push the result into the *greens** array:

```
  things.forEach((e) => {
    greens.push('Green ' + e);
  });
```

10. Log the result. We should get: 'Green Apples', 'Green Arrow', 'Green Bean', etc.
11. Simplify the arrow function syntax by eliminating the callback argument parentheses as well as the function curly braces:

```
things.forEach(e => greens.push('Green ' + e));
```

Iterating over an array to make a new array of transformed items with the same number of items as in the orginal array can also be done with **map()**.

12. Try making the same "greens", but with **map()** instead of **forEach()**. With map, you set the method call equal to the new array which will store the returned a

```
const greenies = things.map((e) => {
  return 'Green ' + e;
})
console.log(greenies);
```

13. Recall that with map, we can make the code ultra-concise if there is only one argument and one line of logic. Try another version, but with no parentheses around the callback argument, **e**, no **return** keyword, and no function curly braces.

```
const g = things.map(e => 'Green ' + e);
console.log(g);
```

As we see, **forEach** does not return a new array automatically like **map()** does.With **forEach** you have to explicitly build the new array, which we do so using **push()** to add new items to the end of the new array.

forEach instead of map-filter chaining
Let's make "green things" again, but this time restricting our product to those things in the original array that have at least six letters. This requires filtering out the shorter words while transformiing the ones that make the cut. This is a job for map-filter or filter-map chaining. We will start with this before doing same with just forEach.

14. Call **filter()** on the **things** array, setting the call equal to the new array being returned:

```
const greenz = things.filter();
```

15. Pass in the callback function that returns when its condition returns true. The condition is items with a length greater than 5:

```
const greenz = things.filter(e => e.length > 5);
```

16. Log the resulting array, which contains only words of at least 6 letters:

```
  console.log(greenz); // 'Coconut', 'Lantern', 'Monster'
```

17. Chain the map method onto the end, to get Green Energy, Green Lantern, etc.:

```
  const greenz = things.filter(e => e.length > 5).map(e => 'Green ' + e);
  console.log(greenz); // 'Green Coconut', 'Green Lantern', 'Green
Monster'
```

18. Now, let's achieve the same result using just forEach. Since forEach does not return a new array automatically, declare a new, empty array as a first step. This is where we will have to push what we make inside the callback function:

```
    const geez = [];
    things.forEach();
```

19. In the forEach callback function, check to see if the current item meets the condition of having a length of at least 6:

```
    things.forEach(e => {
      if(e.length > 5) {
      }
    });
```

20. Inside the if-statement, push the new "green item" into the new array:

```
    things.forEach(e => {
      if(e.length > 5) {
        geez.push('Green ' + e);
      }
    });
```

21. Since the if-statement has just one line of logic, we can omit its curly braces:

```
    things.forEach(e => {
      if(e.length > 5) geez.push('Green ' + e);
    });
```

Using forEach on an array of objects

In this forEach example, we will revisit the Sortable Movies project, where we added new properties to an array of movie objects. In the original project, we added the new properties by of a for loop. We will revisit that approach and then contrast it with **forEach**. The properties we want to add are:

- **hrMin**: an alternative total minutes, where 134 mins is "2h 14m"
- **noAThe**: a version of each name that omits any leading article ("A" or "The")

```
const movies = [
  { name: "The Wizard of Oz", year: 1939, mins: 101 },
  { name: "Casablanca", year: 1942, mins: 102 },
  { name: "Roman Holiday", year: 1953, mins: 118 },
  { name: "Lawrence of Arabia", year: 1962, mins: 227 },
  { name: "Lilies of the Field", year: 1963, mins: 94 },
  { name: "The Godfather, Part II", year: 1974, mins: 202 },
  { name: "The Shining", year: 1980, mins: 144 },
  { name: "Beverly Hills Cop", year: 1984, mins: 105 },
  { name: "Ghost", year: 1990, mins: 127 },
  { name: "Forrest Gump", year: 1994, mins: 142 },
  { name: "Ali", year: 2001, mins: 157 },
  { name: "Due Date", year: 2010, mins: 95 },
  { name: "Black Panther", year: 2018, mins: 134 },
  { name: "Avengers: Endgame", year: 2019, mins: 181 },
];
```

22. First, let's run the for loop version that we recall from the original project:

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

  let mins = movies[i].mins; // get the minute property

  // make a new property in hour-minute format
  movies[i].hrMin = `${Math.floor(mins/60)}h ${mins%60}m`;

  let name = movies[i].name; // get the name property

  // make the noAThe property by checking if the name starts with "A" or
"The"
  // and then making a version of the name that omits the leading
article

  let newName; // newName starts off equal to the original name

  if(name.slice(0,2) == "A ") { // if movie name starts with "A "
    // make the new noAThe property without the "A "
    movies[i].noAThe = name.slice(2);
  } else if(name.slice(0,4) == "The ") { // if movie name starts with
"The "
    // make the new noAThe property without the leading "The "
    movies[i].noAThe = name.slice(4);
```

```
      } else {
        // keep the movie name the same since it does not start with "A " or
  "The "
        movies[i].noAThe = name;
      }

    };

    console.log(movies);
```

23. Now, refactor the above code using **forEach**, with no loop and with the current item as **e**, rather than **movies[i]**:

```
    movies.forEach(e => {

        let mins = e.mins;
        e.hrMin = `${Math.floor(mins/60)}h ${mins%60}m`;
        let name = e.name;
        let newName;

        if(name.slice(0,2) == "A ") {
          e.noAThe = name.slice(2);
        } else if(name.slice(0,4) == "The ") {
          e.noAThe = name.slice(4);
        } else {
          e.noAThe = name;
        }

    });

    console.log(movies);
```

find() method
The **find()** method serves a narrowly specific role: it find the first item in an array that meets a condition, and it returns that item, *only*. It does not look for any more items that meet the condition.

- If NO item that satisfies the is found, the **find()** method returns **undefined**.

First, let's emulate the "find" algorithm using a for loop.

24. Given this **digits** array, set up a for loop that iterates over the array:

```
    let digits = [30, 54, 72, 89, 110, 137, 189];

    for(let i = 0; i < digits.length; i++) {

    }
```

25. Each time through the loop, pass the current item to an if-statement that uses the modulus operator to see if the current number, **digits[i]**, divided by 2 yields a remainder of 1:

```
for(let i = 0; i < digits.length; i++) {
  if(digits[i] % 2 == 1) {
  }
}
```

26. If the condition returns true, the number is odd, so log the odd number, and **break** to quit the loop:

```
for(let i = 0; i < digits.length; i++) {
  if(digits[i] % 2 == 1) {
    console.log(digits[i]);
  }
}
```

27. If we find the target, end the loop with a **break** command. This is especially important if there is a long array with lots of numbers yet to check. We are only finding the first odd number, so there is no point in checking the rest of the array:

```
for(let i = 0; i < digits.length; i++) {
  if(digits[i] % 2 == 1) {
    console.log(digits[i]);
    break;
  }
}
```

28. If no odd number was ever found, log **undeefined**. This must be done only *after* the loop ends and all items in the array have been checked:

```
for(let i = 0; i < digits.length; i++) {
  if(digits[i] % 2 == 1) {
    console.log(digits[i]);
    break;
  }
}
console.log(undefined);
```

The above always logs *undefined*--even if an odd number is found. What we need is a way to log *undefined* only if no odd number is found. We can do this by declaring a boolean **odd = false**, which we toggle to true inside the if-statement. The boolean value is then used as a condition for logging **undefined**:

29. Above the loop, declare a boolean:

```
let odd = false;
```

30. Inside the if statement, flip the boolean to **true**. This must come *before* the break line that ends the loop:

```
  for(let i = 0; i < digits.length; i++) {
    if(digits[i] % 2 == 1) {
      console.log(digits[i]);
      odd = true;
      break;
    }
  }
```

31. Finally, *after* the loop ends, add an if-statement that logs **undefined** if **odd** has stayed false throughout:

```
  if(odd == false) {
    console.log(undefined);
  }
```

32. Change the array values so that there are no odd numbers, to make sure that the **find()** method returns **undefined**.

33. Refactor the if-statement at the end:

- Use the **!** (not operator) instead of **== false**.
- Omit the curly braces, since the if-statement contains only one line of logics

```
    if(!odd) console.log(undefined);
```

find() method instead of for loop
Now let's use the **find()** method to hunt for the first odd number:

34. Call the **find()** method on the **digits** array. Since the method returns a value, set the call equal to a variable to store the return value:

```
  let foundItem = digits.find();
```

35. Add the callback function. Its only code is the condition we are checking. Since it is the callback's only line, omit the function curly braces:

```
    let foundItem = digits.find(e => e % 2 == 1);
    console.log(foundItem);
```

array callback functions use loops under the hood
As we can see, the **find()** method is much more concise than a loop with if-statement. But it is important to understand the process terms of a loop, because the array callback functions -- **map()**, **filter()**, **forEach**, **find()** -- actually use loops "under the hood".

findIndex()
To return the position of the first instance of a target value, use **findIndex()** rather than **find()**.

33. Use **findIndex()** to find the position of the first odd number in the digits array:

```
    let foundIndex = digits.findIndex(e => e % 2 == 1);
    console.log(foundIndex);
```

29. Make sure that the **findIndex()** method returns **-1** by changing the **digits** array values so that there are no odd numbers.

END Lesson 07.03
NEXT: Lab 07.03
Lesson 07.04