Lesson:

Foreach, map, filter, reduce

PW SKILLS

Topics

1. forEach.
2. Introduction to map
3. Introduction to filter.
4. “Break” with forEach, map, and filter.
5. Introduction to reduce.
6. Features of reduce.

Introduction to forEach.

Earlier whenever there was a necessity of iterating over arrays, developers majorly used for loops. But, using loops can sometimes be slow and error-prone when we deal with large complex arrays. So, forEach was standardized with the release of ECMAScript 5 (ES5) in 2009.

The introduction of the forEach() method in ES5 provided a more convenient and efficient way to iterate over the elements of an array. It allowed developers to perform a specific function on each element of an array without having to manually write for a loop. One more thing to note is that within forEach we cannot use the loop control statements such as break, and continue.

Syntax:

```
array.forEach(callback(item, index, array) ⇒ Statements);
```

The following are the parameters of forEach() in Javascript:

- **callbackFunction:** This argument contains the method that will be called for each array element. It is a mandatory parameter.
- **item:** the array item on iteration.
- **index:** This parameter is optional and contains the index number of each element.
- **array:** This parameter is optional and contains the entire array on which the method is being applied.

The return value of this method is always undefined.

Implementation

Let's imagine you are assigned a task to display all the items added to the cart in an e-commerce application. You can do it in multiple ways but let's assume we are asked to do it using forEach.

```
let itemsInCart = [
  "apple",
  "comb",
  "mike",
  "keyboard",
  "t-shirt",
  "mobile holder",
];

// Display the items in cart
itemsInCart.forEach((item) ⇒ console.log(item));
```

Output:

```
/* OUTPUT :
        apple
        comb
        mike
        keyboard
        t-shirt
        mobile holder
*/
```

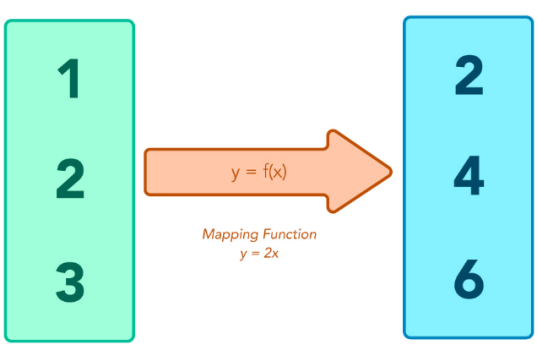
Now let's try to use all the parameters. Let's print the item name, the position in which the item was added to the cart, and the total cart items.

```
itemsInCart.forEach((item, index, arr) ⇒
  console.log(
    `The item ${item} was added to cart in position ${
      index + 1
    }. The items in cart are ${arr}.`
  )
);
```

```
/*

OUTPUT:

The item apple was added to cart in position 1. The items in cart
are apple,comb,mike,keyboard,t-shirt,mobile holder.
The item comb was added to cart in position 2. The items in cart
are apple,comb,mike,keyboard,t-shirt,mobile holder.
The item mike was added to cart in position 3. The items in cart
are apple,comb,mike,keyboard,t-shirt,mobile holder.
The item keyboard was added to cart in position 4. The items in
cart are apple,comb,mike,keyboard,t-shirt,mobile holder.
The item t-shirt was added to cart in position 5. The items in cart
are apple,comb,mike,keyboard,t-shirt,mobile holder.
The item mobile holder was added to cart in position 6. The items
in cart are apple,comb,mike,keyboard,t-shirt,mobile holder.

*/
```

In this case, for each element in the itemsInCart array, the function will log a string to the console. The string includes the current item, its index (incremented by 1 as the indexing starts from 0, to make it more readable), and the full array.

Introduction to map method



The map method in javascript was introduced to iterate and perform operations on array items in a more concise and readable way.

The map method of the array takes a function as a parameter and returns a new array that contains the result of the function performed on each array item. Hence, the new array would be of the same length.

The map() method is used over other iterative methods such as for loops or forEach() based on the applications. map() allows us to easily operate on the array items. It returns a new array containing the result of these operations. As it creates a new array, the array on which the map method is applied would be unaltered and less prone to error.

Syntax:

The map method can be applied to an array in the following ways:

1. Function declaration.
2. Arrow Function.
3. Callback Function.

```
// Function Declaration

array.map(function (item, index, array) {/* Function Body */})

// Arrow Function

array.map((item, index, array) => {/* Function Body */})

// Call back Function

array.map(callback)
```

- item: It is a required parameter and it holds the value of the current item in the iteration.
- index: It is an optional parameter and it holds the index of the current item in the iteration.
- arr: It is an optional parameter and it holds the array.

The return value is a new array whose items are the result of the function performed on each array item.

Implementation

Let's now look at a use case of the map method. Imagine you are working at an e-commerce site. Imagine the website is storing the price as strings and you are assigned a task to convert all the prices in the cart to numbers so the total can be calculated.

```
// Array of item prices as strings
let cartItemsPriceAsStrings = ["100", "58.50", "134", "175", "146",
"205"];

// Array of item prices as numbers
let cartItemsPriceAsNumbers = cartItemsPriceAsStrings.map((item)
⇒ {
return Number(item)
}
);

console.log(cartItemsPriceAsNumbers);


// OUTPUT
[100, 58.5, 134, 175, 146, 205]
```

The above example has an array "cartItemsPriceAsStrings" that contains the prices of items as strings. We will use the map method over that array. Using the "Number()" function to convert each string to a number. The result of each iteration on array items is assigned to a new array called "cartItemsPriceAsNumbers"

Note: If you forget to return a value within the callback function of the “map” method in JavaScript, the resulting array will contain undefined values in the corresponding positions. This happens because “map” creates a new array by applying the callback function to each element of the original array and uses the returned value to populate the corresponding position in the new array.

Introduction to filter method

As the name suggests the filter method is used to filter the data present in an array. On a daily basis in every application, we use filtering. We filter the products in a shopping site based on criteria, we filter the posts on any social media based on the date posted or based on the keywords. So, filtering the data is one of the important task.

The filter method is used to create a new array whose items are the result of the filtering criteria of the original array. Only those values which satisfy the given criteria are added to a new array, and that array is returned. The original array does not get changed.

Whenever the filter method is applied to an array, we pass a filter function to it. The filter function iterates over all the elements of the given array and passes each element to the callback function. If the callback function returns true, then the element is added to the result array otherwise it is ignored.

Syntax:

The callback function can be passed to the filter function in the following ways:

1. Function declaration.
2. Arrow Function.
3. Callback Function.

```
// Function Declaration

array.filter(function (item, index, array) {/* Function Body */})

// Arrow Function

array.filter((item, index, array) => {/* Function Body */})

// Call back Function

array.filter(callback)
```

- item: It is a required parameter and it holds the value of the current item in the iteration.
- index: It is an optional parameter and it holds the index of the current item in the iteration.
- arr: It is an optional parameter and it holds the array.

The filter function returns an array. if the callback function returns truthy, element is added in the output array and if the callback function returns falsy, then that element is removed in the output array. It does not affects the original array.

Implementation

Let's imagine you are working for an e-commerce organization and you are assigned to return the long usernames from the data of the username. Long usernames in this case are usernames whose length is greater than 5 characters.

```
let userNames = ["Nasikh", "Anurag", "Alka", "Prabir", "Vinay"];

let longUserNames = userNames.filter((item) ⟹ item.length > 5);
console.log(longUserNames); // OUTPUT : [ 'Nasikh', 'Anurag',
'Prabir' ]
```

The filter() method creates a new array "longUserNames" by filtering out the names with a length greater than or equal to 5.

The filter() method iterates through each element in the userNames array, and for each element, it calls the provided function ((item) => item.length > 5) with the element as the argument.

If the function returns true for that element (if the length of the name is greater than 5), it is included in the longUserNames array, otherwise, it is excluded.

"Break" with forEach, map and filter

The "break" keyword does not work in the forEach, map, and filter methods in JavaScript like it does in traditional loops such as for or while loops. The "forEach", "map", and "filter" methods are higher-order functions that operate on arrays and execute a provided callback function for each element. These methods do not provide a direct mechanism for breaking out of the iteration process. So these methods have their own internal control flow, and the iteration behavior is predefined.

If we write "break" keyword along with "forEach", "map", or "filter", it will throw us Syntax error

Example for the syntax error:

```
const numbers = [1, 2, 3, 4, 5];
```

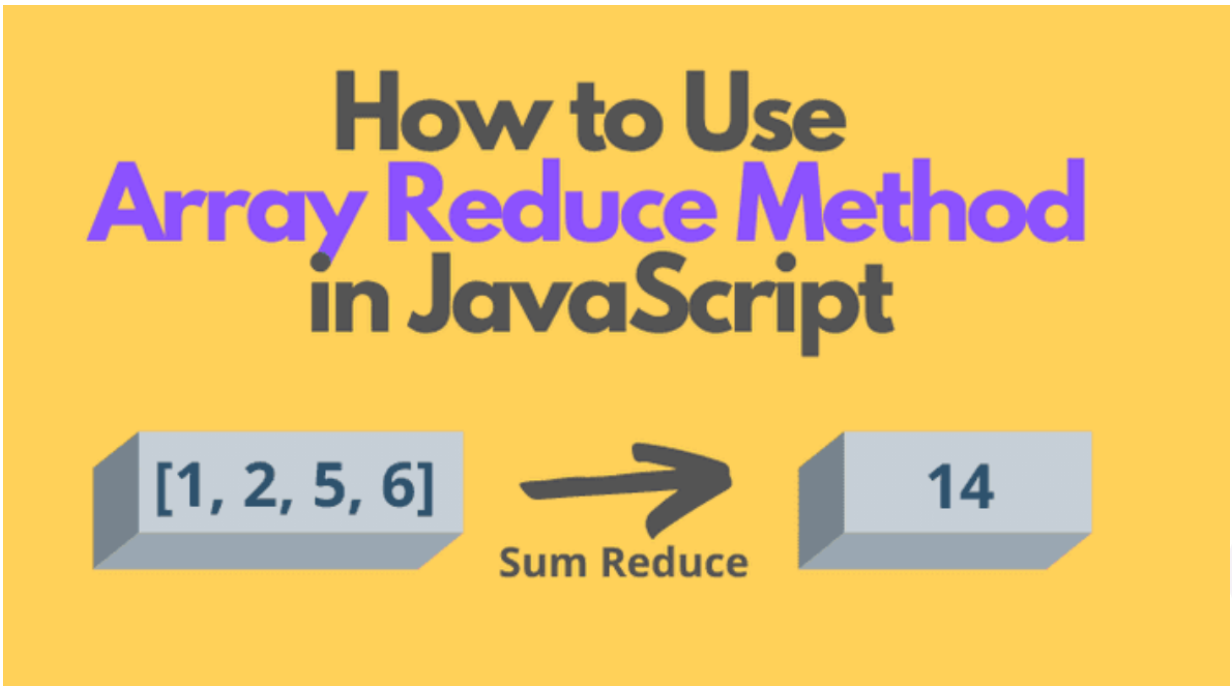
```
// Using forEach
numbers.forEach(number => {
  if (number === 3) {
```

```
    break; // This will throw an error
  }
  console.log(number);
});
// Output: Uncaught SyntaxError: Illegal break statement
```

```
// Using map
const mapped = numbers.map(number => {
  if (number === 3) {
    break; // This will throw an error
  }
  return number * 2;
});
// Output: Uncaught SyntaxError: Illegal break statement
```

```
// Using filter
const filtered = numbers.filter(number => {
  if (number === 3) {
    break; // This will throw an error
  }
  return true;
});
// Output: Uncaught SyntaxError: Illegal break statement
```

Introduction to reduce.



Introduction

As the name suggests the reduce method reduces the array values to a single value. The reduce() method runs a reducer function on each array item and returns a single output result. The input array can have numbers, strings, or an object.

Features of reduce

1. The reducer function passed will be applied to all the items in the array.

2. The result will be a single value accumulated on passing all the array items to the reducer function.

3. The reduce method is not valid for empty arrays.

4. The reduce method doesn't change the original array on which the reduce method is applied.

Syntax:

```
array.reduce(function(accumulator, currentValue, currentIndex, array), initialValue)
```

The parameters accepted:

1. accumulator: It accumulates the initial value and reducer function's return values.

2. currentValue: It is a required parameter and is used to specify the value of the current element.

3. currentIndex: It is an optional parameter and is used to specify the array index of the current element.

4. array: It is an optional parameter and is used to specify the array the current element belongs to.

5. initialValue: It is used to specify the value to be passed to the function as the initial value. It's optional.

Implementation

Let's look at the implementation of the reduce method. The simplest example to demonstrate the use of reduce function is by finding the sum of all elements in an array.

Find the sum of all elements in an array?

```
// Given Array

let arr = [1, 2, 44, 67, 89]
```

```
// Apply reduce method

let result = arr.reduce((acc, curr) ⇒ acc + curr, 0);

// Print the result to the console.
console.log(result);


// output
203
```

We are passing in a callback function as the reduce function (acc, curr) => acc + curr, which takes in the accumulator and current value as arguments, and adds the current value to the accumulator. We have also passed in the initial value as 0, which is the starting value of the accumulator.

Note: If the initial value is not passed to the reduce method, the first item in the sequence is used as the initial value and the operation starts from the second item in the sequence. If the sequence is empty, a TypeError is raised.

Example1:

```
// Given Array

let arr = [4, 4, 2]


// Apply reduce method

let result = arr.reduce((acc, curr) ⇒ acc + curr);
//here initial value is take as 4

// Print the result to the console.
console.log(result);

// output

10
```

Example2:

```
const emptyArray = [];

const sum = emptyArray.reduce(function(acc, num) {
  return acc + num;
});
console.log(sum); // Raises TypeError: Reduce of empty array with
no initial value
```

The reduce method iterates through the array and for each element, it applies the callback function and updates the accumulator, adding the current value to the accumulator and storing the result in the accumulator.

After iterating through the entire array, the final value of the accumulator is returned, which is the sum of all elements in the array.

Now lets have a look at how we can use reduce function on an array of objects

```
const students = [
  { name: 'Alice', score: 85 },
  { name: 'Bob', score: 90 },
  { name: 'Charlie', score: 75 },
  { name: 'Dave', score: 95 }
];

const highestScoringStudent = students.reduce(function(acc,
student) {
  if (student.score > acc.score) {
    return student;
  } else {
    return acc;
  }
});
console.log(highestScoringStudent); // Output: { name: 'Dave',
score: 95 }
```

In this example, we have an array of student objects, where each object contains the properties name and score. We use the reduce method to find the student with the highest score in the array.

The callback function passed to reduce takes two arguments: acc (accumulator) and student (current object in the iteration). In this case, the accumulator initially holds the first student object in the array.

In each iteration, the callback function compares the score of the current student with the score of the accumulator. If the score of the current student is higher, the current student object is returned and becomes the new accumulator. Otherwise, the accumulator remains unchanged.

Interview Points

1. Implement a JavaScript function that uses the filter method to extract only the even numbers from an array. Provide an example array and demonstrate the use of your function.

Ans.

```
function filterEvenNumbers(arr) {
  return arr.filter((item) => item % 2 === 0);
}

const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
console.log(filterEvenNumbers(numbers)); // Output should be an
array with only // Output - [2, 4, 6, 8, 10]
```

2. Compare and contrast the **map** and **forEach** array method. Can you give an example of a practical scenario where the map would be advantageous over other looping methods?

forEach vs map

3. Imagine you have an array of objects representing products and need to extract an array of just the product names. Here's where the map can be advantageous:

```
const products = [
  { id: 1, name: 'Laptop', price: 999 },
  { id: 2, name: 'Phone', price: 499 },
  { id: 3, name: 'Tablet', price: 299 }
];

// Using map to extract an array of product names
const productNames = products.map(product ⇒ product.name);
// productNames: ['Laptop', 'Phone', 'Tablet']
```

In this scenario, the map is advantageous because it allows you to create a new array containing only the product names in a concise and readable manner. If you were to use forEach for the same task, you would need to declare an empty array, push each product name into it, and manage the array manually. map simplifies this process by handling the creation of the new array for you.