Hema Coding School

YouTube Link: https://www.youtube.com/@HemaCodingSchool

Higher-Order Functions in JavaScript

Definition:

A higher-order function is a function that does at least one of the following:

1. Takes one or more functions as arguments.
2. Returns a function as its result.

Higher-order functions allow for more abstract and flexible ways to manipulate and control other functions and their behavior.

```
let result = (val) => {
  console.log(`Call back Result...${val}`);
};

let add = (a, b, callback) => {
  let c = a + b;
  callback(c)
    // console.log(a + b);
  return () => console.log(`Return function ${c}`);
};

let x = add(1, 2,result);
x()
```

Map():

The map() method in JavaScript is a higher-order function that is used to transform elements of an array. It iterates over each element in the array, applies a callback function to each element, and returns a new array with the results of applying the callback function to each element.

```
array.map(callback(currentValue[, index[, array]]) {
    // return element for newArray, after executing something
  }[, thisArg]);
```

- **callback**: The function to execute on each element of the array.
- **currentValue**: The current element being processed in the array.
- **index** (optional): The index of the current element being processed in the array.
- **array** (optional): The array **map** was called upon.
- **thisArg** (optional): Value to use as **this** when executing **callback**.

The **map()** method does not mutate the original array; instead, it creates a new array with the results of calling the callback function on each element.

Example:

```
//Map()
let originalArray = [1, 2, 3, 4, 5];

let newArray = originalArray.map((currValue, ind, arr) => {

//   console.log(currValue * 2);
    return currValue % 2 === 0
});

console.log(newArray);

let originalArray = [1, 2, 3, 4, 5];
let multiply = {
  number: 2,
};

let newArray = originalArray.map(function(currValue, ind, arr) {
//   console.log(currValue * this.number);
  return currValue * this.number;
}, multiply);

console.log(newArray);
```

Filter():

The filter() method in JavaScript is another higher-order function that is used to create a new array with all elements that pass a certain condition. It iterates over each element in the array, applies a callback function to each element, and includes the element in the new array if the callback function returns true

```
array.filter(callback(element[, index[, array]]) {
    // return true to keep the element in the new array, false otherwise
  }[, thisArg]);
```

- **callback**: The function to test each element of the array.
- **element**: The current element being processed in the array.
- **index** (optional): The index of the current element being processed in the array.
- **array** (optional): The array **filter** was called upon.
- **thisArg** (optional): Value to use as **this** when executing **callback**.

The **filter()** method creates a new array containing only the elements that pass the test implemented by the callback function.

```
//Filter()
```

```
let originalArray = [1, 2, 3, 4, 5];

let filterArray = originalArray.filter((currValue, ind, arr) => {
  return currValue % 2 === 0;
});

console.log(filterArray);
```

Reduce():

The reduce() method in JavaScript is a higher-order function that applies a callback function (known as the reducer function) against an accumulator and each element in an array, from left to right, to reduce it to a single value. It is particularly useful when you want to perform some computation on all elements of an array and then return a single value based on that computation.

```
array.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])
```

- **callback**: A function to execute on each element in the array, taking four arguments:
- **accumulator**: The accumulator accumulates the callback's return values. It is the accumulated value previously returned in the last invocation of the callback, or **initialValue**, if supplied (see below).
- **currentValue**: The current element being processed in the array.
- **index** (optional): The index of the current element being processed in the array.
- **array** (optional): The array **reduce()** was called upon.
- **initialValue** (optional): A value to use as the first argument to the first call of the **callback**. If no initial value is supplied, the first element in the array will be used as the initial accumulator value, and the callback will be invoked for the second element onwards.

Example:

```
let originalArray = [6, 1, 2, 3, 4, 9, 5, 6, 8, 7];

let reduceArray = originalArray.reduce((acc, currValue, ind, arr) => {
    return acc + currValue;
});

console.log(reduceArray);
```

ForEach():

The forEach() method in JavaScript is another higher-order function used to iterate over elements in an array. It executes a provided function once for each array element and does not return a new array. Its primary purpose is to perform a side effect for each element in the array.

```
array.forEach(callback(currentValue[, index[, array]])[, thisArg])
```

- **callback**: A function to execute for each element in the array, taking three arguments:
- **currentValue**: The current element being processed in the array.
- **index** (optional): The index of the current element being processed in the array.
- **array** (optional): The array **forEach()** was called upon.
- **thisArg** (optional): A value to use as **this** when executing the callback function.

Example:

```
//ForEach():
const fruits = ["apple", "banana", "cherry"];

// Log each fruit to the console using forEach
fruits.forEach((fruit, index) => {
   console.log(`Fruit at index ${index}: ${fruit}`);
});
```

Include():

The includes() method is used to determine whether an array includes a certain element, returning true or false as appropriate. It checks if an element is present in the array and returns true if found, false otherwise.

```
array.includes(searchElement, fromIndex)
```

- **array**: The array to search for the specified element.
- **searchElement**: The element to search for within the array.
- **fromIndex** (optional): The index at which to start the search. If this value is negative, it is treated as **array.length + fromIndex**. If **fromIndex** is greater than or equal to the array's length, **includes()** returns **false**. Default is **0**.

Example:

```
const fruits = ['apple', 'banana', 'cherry', 'date'];

console.log(fruits.includes('banana'));  // Outputs: true
console.log(fruits.includes('grape'));   // Outputs: false

// Starting search from index 2
console.log(fruits.includes('cherry', 2)); // Outputs: true
```

```
// Starting search from index 3 (negative index)
console.log(fruits.includes('banana', -3)); // Outputs: true

// Starting search from index 5 (out of bounds)
console.log(fruits.includes('banana', 5)); // Outputs: false
```

Every():

The every() method in JavaScript is used to test whether all elements in an array pass the test implemented by the provided function. It returns true if all elements in the array pass the test; otherwise, it returns false.

```
array.every(callback(currentValue[, index[, array]])[, thisArg])
```

- **array**: The array to be tested.
- **callback**: A function to test for each element, taking three arguments:
- **currentValue**: The current element being processed in the array.
- **index** (optional): The index of the current element being processed in the array.
- **array** (optional): The array **every()** was called upon.
- **thisArg** (optional): A value to use as **this** when executing the callback function

Example:

```
const numbers = [2, 4, 6, 8, 10];

// Check if all numbers are even
const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // Outputs: true

// Check if all numbers are greater than 5
const greaterThanFive = numbers.every(num => num > 5);
console.log(greaterThanFive); // Outputs: false
```

Some():

The some() method in JavaScript is used to test whether at least one element in the array passes the test implemented by the provided function. It returns true if at least one element in the array passes the test; otherwise, it returns false.

```
array.some(callback(currentValue[, index[, array]])[, thisArg])
```

- **array**: The array to be tested.

- **callback**: A function to test for each element, taking three arguments:
- **currentValue**: The current element being processed in the array.
- **index** (optional): The index of the current element being processed in the array.
- **array** (optional): The array **some()** was called upon.
- **thisArg** (optional): A value to use as **this** when executing the callback function.

Example:

```
const numbers = [1, 3, 5, 7, 9];

// Check if some numbers are even
const someEven = numbers.some(num => num % 2 === 0);
console.log(someEven); // Outputs: false

// Check if some numbers are greater than 5
const someGreaterThanFive = numbers.some(num => num > 5);
console.log(someGreaterThanFive); // Outputs: true
```

Interview Questions:

What is a higher-order function in JavaScript?

Answer: A higher-order function is a function that either takes one or more functions as arguments or returns a function as its result.

Can you provide an example of a higher-order function that takes another function as an argument?

```
 function greet(name) {
   console.log(`Hello, ${name}!`);
}

function processUserInput(callback) {
   const name = prompt('Please enter your name.');
   callback(name);
}

processUserInput(greet);
```

How do higher-order functions contribute to functional programming?

Answer: Higher-order functions allow for greater abstraction and code reuse. They enable functions to be composed and combined, making it easier to build complex operations from simpler ones.

What are some common use cases for higher-order functions?

Answer: Common use cases include callbacks for event handling, array methods like map, filter, and reduce, and creating function factories or decorators.

map

What does the map method do in JavaScript?

Answer: The map method creates a new array populated with the results of calling a provided function on every element in the calling array.

What is the difference between map and forEach?

Answer: The map method returns a new array with the results of calling a function on every element, whereas forEach executes a function on each element but does not return a new array.

Can map change the original array?

Answer: No, map does not modify the original array. It returns a new array with the transformed elements.

filter

What does the filter method do in JavaScript?

Answer: The filter method creates a new array with all elements that pass the test implemented by the provided function.

What is the difference between filter and map?

Answer: The filter method creates a new array with elements that pass a test, whereas map transforms each element in the array and returns a new array with the transformed elements.

Can filter change the original array?

Answer: No, filter does not modify the original array. It returns a new array with the elements that pass the test.

reduce

What does the reduce **method do in JavaScript?**

Answer: The reduce method executes a reducer function on each element of the array, resulting in a single output value.

What are the parameters of the callback function used in reduce?

Answer: The callback function in reduce takes four arguments:

1. **accumulator:** The accumulator accumulates the callback's return values.
2. **currentValue:** The current element being processed in the array.
3. **currentIndex (optional):** The index of the current element being processed in the array.
4. **array (optional):** The array reduce was called upon.

Can reduce change the original array?

Answer: No, reduce does not modify the original array. It reduces the array to a single value based on the provided function.

What is the advantage of using map, filter, and reduce over traditional for loops?

- **Answer:** These methods provide a more declarative and readable way to process arrays. They also support functional programming principles, making the code more concise and easier to understand.