Understanding Higher-Order Functions in JavaScript

Learn What Higher-Order Functions are and how to use them in JavaScript

Sukhjinder Arora Follow
Oct 23, 2018 · 7 min read

Photo by NESA by Makers on Unsplash

If you are learning JavaScript, you must have come across the term *Higher-Order Functions.* Although it may sound complicated, it isn't.

What makes JavaScript suitable for functional programming is that it accepts Higher-Order Functions.

Higher-Order Functions are extensively used in JavaScript. If you have been programming in JavaScript for a while, you may have already used them without even knowing.

To fully understand this concept, you first have to understand what *Functional Programming* is and the concept of *First-Class Functions*.

Tip: Use **Bit (GitHub)** to easily share and reuse components between apps, to build faster as a team. It works with any JS code, so feel free to give it a go.

React spinners with Bit

What is Functional Programming

In most simple term, Functional Programming is a form of programming in which you can pass functions as parameters to other functions and also return them as values. In functional programming, we think and code in terms of functions.

JavaScript, Haskell, Clojure, Scala, and Erlang are some of the languages that implement functional programming.

First-Class Functions

If you have been learning JavaScript, you may have heard that JavaScript treats functions as first-class citizens. That's because in JavaScript or any other functional programming languages functions are objects.

In JavaScript functions are a special type of objects. They are `Function` objects. For example:

```
function greeting() {
  console.log('Hello World');
}
// Invoking the function
greeting();  // prints 'Hello World'
```

To prove functions are objects in JavaScript, we could do something like this:

```
// We can add properties to functions like we do with objects
greeting.lang = 'English';
// Prints 'English'
console.log(greeting.lang);
```

Note — While this is perfectly valid in JavaScript, this is considered a harmful practice. You should not add random properties to the function objects, use an object if you have to.

In JavaScript, everything you can do with other types like object, string, or number, you can do with functions. You can pass them as parameters to other functions (callbacks), assign them to variables and pass them around etc. This is why functions in JavaScript are known as First-Class Functions.

Assigning Functions to Variables

We can assign functions to variables in JavaScript. For example:

```
const square = function(x) {
  return x * x;
}
// prints 25
square(5);
```

We can also pass them around. For example:

```
const foo = square;
// prints 36
foo(6);
```

```
  console.log("What's up?");
}
function greet(type, greetFormal, greetCasual) {
  if(type === 'formal') {
    greetFormal();
  } else if(type === 'casual') {
    greetCasual();
  }
}
// prints 'What's up?'
greet('casual', formalGreeting, casualGreeting);
```

Now that we know what first-class functions are, let's dive into Higher-Order functions in JavaScript.

Higher-Order Functions

Higher order functions are functions that operate on other functions, either by taking them as arguments or by returning them. In simple words, A Higher-Order function is a function that receives a function as an argument or returns the function as output.

For example, `Array.prototype.map` , `Array.prototype.filter` and `Array.prototype.reduce` are some of the Higher-Order functions built into the language.

Higher-Order Functions in Action

Let's take a look at some examples of built-in higher-order functions and see how it compares to solutions where we aren't using Higher-Order Functions.

Array.prototype.map

The `map()` method creates a new array by calling the callback function provided as an argument on every element in the input array. The `map()` method will take every returned value from the callback function and creates a new array using those values.

The callback function passed to the `map()` method accepts 3 arguments: `element` , `index` , and `array` .

Let's look at some examples:

Example 1#

Let's say we have an array of numbers and we want to create a new array which contains double of each value of the first array. Let's see how we can solve the problem with and without Higher-Order Function.

Without Higher-order function

```
const arr1 = [1, 2, 3];
const arr2 = [];
for(let i = 0; i < arr1.length; i++) {
  arr2.push(arr1[i] * 2);
}
// prints [ 2, 4, 6 ]
console.log(arr2);
```

With Higher-order function `map`

```
const arr1 = [1, 2, 3];
const arr2 = arr1.map(function(item) {
  return item * 2;
});
console.log(arr2);
```

We can make this even shorter using the arrow function syntax:

```
const arr1 = [1, 2, 3];
const arr2 = arr1.map(item => item * 2);
console.log(arr2);
```

Example 2#

Let's say we have an array containing the birth year of different persons and we want to create an array that contains their ages. For example:

Without Higher-order function

```
// prints [ 43, 21, 16, 23, 33 ]
console.log(ages);
```

With Higher-order function `map`

```
const birthYear = [1975, 1997, 2002, 1995, 1985];
const ages = birthYear.map(year => 2018 - year);
// prints [ 43, 21, 16, 23, 33 ]
console.log(ages);
```

Array.prototype.filter

The `filter()` method creates a new array with all elements that pass the test provided by the callback function. The callback function passed to the `filter()` method accepts 3 arguments: `element`, `index`, and `array`.

Let's look at some examples:

Example 1#

Let's say we have an array which contains objects with name and age properties. We want to create an array that contains only the persons with full age (age greater than or equal to 18).

Without Higher-order function

```
const persons = [
  { name: 'Peter', age: 16 },
  { name: 'Mark', age: 18 },
  { name: 'John', age: 27 },
  { name: 'Jane', age: 14 },
  { name: 'Tony', age: 24},
];
const fullAge = [];
for(let i = 0; i < persons.length; i++) {
  if(persons[i].age >= 18) {
    fullAge.push(persons[i]);
  }
}
console.log(fullAge);
```

With Higher-order function `filter`

```
const persons = [
  { name: 'Peter', age: 16 },
  { name: 'Mark', age: 18 },
  { name: 'John', age: 27 },
  { name: 'Jane', age: 14 },
  { name: 'Tony', age: 24},
];
const fullAge = persons.filter(person => person.age >= 18);
console.log(fullAge);
```

Array.prototype.reduce

The `reduce` method executes the callback function on each member of the calling array which results in a single output value. The reduce method accepts two parameters: 1) The reducer function (callback), 2) and an optional `initialValue`.

The reducer function (callback) accepts four parameters: `accumulator`, `currentValue`, `currentIndex`, `sourceArray`.

If an `initialValue` is provided, then the `accumulator` will be equal to the `initialValue` and the `currentValue` will be equal to the first element in the array.

If no `initialValue` is provided, then the `accumulator` will be equal to the first element in the array and the `currentValue` will be equal to the second element in the array.

Example 1#

Let's say we have to sum an array of numbers:

With Higher-order function `reduce`

```
const arr = [5, 7, 1, 8, 4];
const sum = arr.reduce(function(accumulator, currentValue) {
  return accumulator + currentValue;
});
```

Top highlight

We can also provide an initial value to this function:

```
const arr = [5, 7, 1, 8, 4];
const sum = arr.reduce(function(accumulator, currentValue) {
  return accumulator + currentValue;
}, 10);
// prints 35
console.log(sum);
```

Without Higher-order function

```
const arr = [5, 7, 1, 8, 4];
let sum = 0;
for(let i = 0; i < arr.length; i++) {
  sum = sum + arr[i];
}
// prints 25
console.log(sum);
```

You can see that using High-order function made our code cleaner, more concise and less verbose.

Creating Our own Higher-Order Function

Up until this point, we saw various Higher-order functions built into the language. Now let's create our own Higher-order function.

Let's imagine JavaScript didn't have the native map method. We could build it ourselves thus creating our own Higher-Order Function.

Let's say, we have an array of strings and we want to convert this array to an array of integers, where each element represent the length of the string in the original array.

```
const strArray = ['JavaScript', 'Python', 'PHP', 'Java', 'C'];
function mapForEach(arr, fn) {
  const newArray = [];
  for(let i = 0; i < arr.length; i++) {
    newArray.push(
      fn(arr[i])
    );
  }
  return newArray;
}
const lenArray = mapForEach(strArray, function(item) {
  return item.length;
});
// prints [ 10, 6, 3, 4, 1 ]
console.log(lenArray);
```

In the above example, we created an Higher-order function `mapForEach` which accepts an array and a callback function `fn` . This function loops over the provided array and calls the callback function `fn` inside the `newArray.push` function call on each iteration.

The callback function `fn` receives the current element of array and returns the length of that element, which is stored inside the `newArray` . After the for loop has finished, the `newArray` is returned and assigned to the `lenArray` .

Conclusion

We have learned what Higher-order functions are and some built-in Higher-order function. We also learned how to create our own Higher-order functions.

In a nutshell, a Higher-order function is a function that may receive a function as an argument and can even return a function. Higher-order functions are just like regular functions with an added ability of receiving and returning other functions are arguments and output.

That's it and if you found this article helpful, please click the clap button, you can also follow me on Medium and Twitter, and if you have any doubt, feel free to comment! I'd be happy to help :)

. . .

Learn more

Understanding Execution Context and Execution Stack in Javascript

Understanding execution context and stack to become a better Javascript developer.

blog.bitsrc.io

11 React UI Component Libraries you Should Know in 2018

11 React component libraries with great components for building your next app's UI interface in 2018.

blog.bitsrc.io

JavaScript Software Development Web Development Programming Nodejs

7.3K claps

WRITTEN BY

Sukhjinder Arora

Follow

Web Developer. Tech Writer. Loves poetry, philosophy and programming. Find me @ https://sukhjinderarora.com/

Bits and Pieces

Follow

Coding in the age of code components. Learn more @ bit.dev

See responses (23)

More From Medium

More from Bits and Pieces

More from Bits and Pieces

More from Bits and Pieces

“In Simple English”

Yann Mulonda in Bits...
Jul 17 · 6 min read ★ 553

Developer

Chris Gregori in Bits...
Jun 17 · 7 min read ★ 1.6K

Developer Should Know

Chidume Nnamdi □...
Oct 9, 2018 · 11 min... 27K