

JavaScript Higher Order Functions

Last Updated : 15 Apr, 2025



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

- Takes another function as an argument.
- Returns another function as its result.

Higher-order functions help make your code more reusable and modular by allowing you to work with functions like any other value.

```
function fun() {  
    console.log("Hello, World!");  
}  
function fun2(action) {  
    action();  
    action();  
}  
  
fun2(fun);
```



In this example

- fun2 is a higher-order function because it takes another function (action) as an argument.
- It calls the action function twice.

Popular Higher Order Functions in JavaScript

1. map

The [map](#) function is used to transform an array by applying a callback function to each element. It returns a new array.

```
const n = [1, 2, 3, 4, 5];  
const square = n.map((num) => num * num);  
console.log(square);
```



- `map` applies the callback `(num) => num * num` to each element of numbers.
- A new array is returned where each element is the square of the original

2. filter

The [filter](#) function is used to create a new array containing elements that satisfy a given condition.

```
const n = [1, 2, 3, 4, 5];  
const even = n.filter((num) => num % 2 === 0);  
console.log(even);
```

- The callback `(num) => num % 2 === 0` filters out elements not divisible by 2.
- The resulting array contains only even numbers.

3. reduce

The `reduce` function accumulates array elements into a single value based on a callback function.

```
const n = [1, 2, 3, 4, 5];  
const sum = n.reduce((acc, curr) => acc + curr, 0);  
console.log(sum);
```

- The callback `(acc, curr) => acc + curr` adds all elements.
- 0 is the initial value of the `acc`.

4. forEach

The [forEach](#) function executes a provided function once for each array element.

```
const n = [1, 2, 3];  
n.forEach((num) => console.log(num * 2));
```

- `forEach` performs the side effect of printing each element multiplied by 2.
- It does not return a new array like `map`.

5. find

The [find](#) function returns the first element in the array that satisfies a given condition.

```
const n = [1, 2, 3, 4, 5];  
const fEven = n.find((num) => num % 2 === 0);  
console.log(fEven);
```

- The callback (num) => num % 2 === 0 finds the first even number.
- If no element satisfies the condition, it returns undefined.

6. some

The [some](#) function checks if at least one array element satisfies a condition.

```
const n = [1, 2, 3, 4, 5];  
const hasNeg = n.some((num) => num < 0);  
console.log(hasNeg);
```

- The callback (num) => num < 0 checks for negative numbers.
- It returns true if any element passes the condition, false otherwise.

7. every

The [every](#) function checks if all array elements satisfy a condition.

```
const n = [1, 2, 3, 4, 5];  
const allPos = n.every((num) => num > 0);  
console.log(allPos)
```

- The callback (num) => num > 0 checks if all numbers are positive.
- It returns true only if all elements pass the condition.

Advanced Techniques with Higher Order Functions

1. Function Composition

[Function composition](#) is the process of combining multiple functions to create a new function. The composed function applies multiple



operations in sequence.

```
function add(x) {  
    return x + 2;  
}  
function mul(x) {  
    return x * 3;  
}  
  
function compose(f, g) {  
    return function(x) {  
        return f(g(x));  
    };  
}  
  
var res = compose(add, mul)(4);  
console.log(res);
```

- compose combines add and multiply, so the output of multiply is passed as input to add.
- The result of `compose(add, mul)(4)` is 14 because 4 is first multiplied by 3 and then 2 is added.

2. Currying

Currying transforms a function that takes multiple arguments into a series of functions that each take one argument. This allows partial application of the function.

```
function mul(x) {  
    return function(y) {  
        return x * y;  
    };  
}  
  
var mul = mul(2);  
console.log(mul(5));
```

- The multiply function is curried, returning a new function each time it is called with an argument.
- `multiplyBy2` is a partially applied function that multiplies any given number by 2.

3. Memoization

Memoization is a technique where function results are cached so that repeated calls with the same arguments return faster. This is particularly useful for expensive function calls.



```
function memoize(func) {
  var cache = {};
  return function (arg) {
    if (cache[arg]) {
      return cache[arg];
    } else {
      var res = func(arg);
      cache[arg] = res;
      return res;
    }
  };
}

function slow(num) {
  console.log("Computing...");
  return num * 2;
}

var fast = memoize(slow);
console.log(fast(5)); // Computing... 10
console.log(fast(5)); // 10 (cached)
```

- memoize caches the results of slowFunction calls. The second time fast(5) is called, the result is fetched from the cache, avoiding recomputation.
- This optimization improves performance by saving on redundant calculations.

Use case's of higher order functions

1. Passing Functions as Arguments

In the following example, we define a Higher-Order Function called greet that accepts a [callback](#) function as an argument and executes it

```
function greet(name, callback) {
  console.log("Hello, " + name);
  callback();
}

function sayGoodbye() {
  console.log("Goodbye!");
}

► {...}
```

- **Function as Argument:** greet accepts another function (e.g., sayGoodbye) as a callback, demonstrating the ability to pass functions as arguments.

- **Sequence Control:** It first logs a greeting message and then executes the callback, showing how actions can be performed in a specific order.
- **Modularity and Reusability:** By separating the greeting and goodbye actions, the pattern allows flexibility and reusability, enabling different callbacks to be passed as needed.

2. Returning Functions from Functions

Higher-order functions can also return a function. This enables the creation of more [dynamic behavior](#)

```
function mul(factor) {  
    return function(num) {  
        return num * factor;  
    };  
}  
  
► {...}
```

- **Function Factory:** mulBy returns a new function based on the provided factor, demonstrating the ability to create dynamic, parameterized functions.
- **Closure in Action:** The returned function uses the captured factor to perform multiplication, showcasing the power of closures to retain access to external variables.
- **Reusability and Customization:** This pattern simplifies creating reusable multipliers (e.g., mul2, mul3), enabling efficient and customizable solutions with minimal effort.

3. Array Method `map()` as a Higher-Order Function

JavaScript [array methods](#) such as [map\(\)](#), [filter\(\)](#), and [reduce\(\)](#) are excellent examples of higher-order functions. These methods take callback functions as arguments and provide powerful ways to manipulate arrays.

```
const a = [1, 2, 3, 4, 5];  
const double = a.map(function(n) {  
    return n * 2;  
});
```

```
console.log(double);
```

- **Array Transformation:** `map()` applies a callback function to each array element, returning a new array with transformed values while keeping the original array unchanged
- **Immutability:** By not mutating the original array, `map()` supports immutable data handling, which is key to predictable and safer code.
- **Declarative Iteration:** It abstracts the iteration logic, promoting a declarative programming style that focuses on what should be done rather than how.

4. Array Method `filter()` as a Higher-Order Function

The [filter\(\) method](#) is another array function that is a higher-order function. It filters the elements of an array based on a condition provided by the callback function.

```
const a = [1, 2, 3, 4, 5];
const even = a.filter(function(n) {
  return n % 2 === 0;
});

console.log(even);
```

- **Conditional Filtering:** `filter()` applies a callback function to test each element, returning a new array containing only those that meet the specified condition.
- **Immutability:** It leaves the original array unchanged, ensuring the integrity of the source data while providing filtered results.
- **Customizable and Reusable:** `filter()` is highly flexible, allowing easy customization for different conditions to extract specific subsets of data.

5. Array Method `reduce()` as a Higher-Order Function

The [reduce\(\)](#) method is a powerful higher-order function used to reduce an array to a single value.

```
const n = [1, 2, 3, 4, 5];
const sum = n.reduce(function(acc, curr) {
  return acc + curr;
});
```



```
}, 0);  
console.log(sum);
```

- **Accumulation:** `reduce()` processes each element of the array, accumulating a single value (e.g., sum, product) based on the provided callback function.
- **Initial Value and Flexibility:** The second argument (e.g., 0) sets the initial value for the accumulator, ensuring consistent results and allowing for flexible aggregation.
- **Versatility:** It can be used for a wide range of tasks, such as summing values, calculating products, or even more complex operations like flattening arrays.

[Comment](#)[More info](#) ▼[Campus Training Program](#)[Next Article >](#)

How to Create a GitHub Profile
Search using HTML CSS and
JavaScript ?

Similar Reads

