- # Higher-Order Functions in JavaScript

  Learning Objectives

  By the end of this lesson, you will understand:

  - The definition and purpose of Higher-Order Functions (HOF).
  - How to use `map()` to transform arrays in JavaScript.
  - How to use `forEach()` for iterating over arrays.
  - The difference between `map()` and `forEach()` in terms of behavior and use cases.
  - How to use `filter()` to create subsets of arrays.
  - How to use `reduce()` to accumulate values into a single result.
  - How to use the `sort()` method to sort arrays.
  - The default behavior of `sort()` and how it compares values.
  - How to provide a custom sorting function for more complex data types.
  - What method chaining is and why it is useful in JavaScript.
  - How to chain Higher-Order Functions (HOFs) like `map()`, `forEach()`, `filter()`, and `reduce()`.

- ---

  1. Higher-Order Functions (HOF) Definition

  What is a Higher-Order Function?

  A Higher-Order Function (HOF) is a function that either:

  - Takes another function as an argument, or
  - Returns a function as its result.

- Higher-order functions are a key aspect of functional programming, allowing for more abstract, reusable, and cleaner code. They are used for handling common patterns, such as iteration, event handling, and data transformation.

  Example 1: HOF Taking a Function as an Argument

```
function applyOperation(a, b, operation) {
  return operation(a, b);
}

function add(x, y) {
  return x + y;
}

console.log(applyOperation(5, 3, add));  // Output: 8
```

- Example 2: HOF Returning a Function

```
function multiplier(factor) {
  return function (number) {
    return number * factor;
  };
}

let double = multiplier(2);
console.log(double(5));  // Output: 10
```

-
- In Example 1, `applyOperation` is a higher-order function because it takes the `add` function as an argument. In Example 2, `multiplier` is a higher-order function because it returns a new function.

---

2. Using `map()`: Transforming Arrays

The `map()` method creates a new array by calling a provided function on every element in the array. The original array is not modified.

Syntax:

```
array.map(callback);
```

-
  `callback`: A function that is called on each element of the array. The function takes three arguments:
  `currentValue`: The current element being processed.
  `index` (optional): The index of the current element.
  `array` (optional): The array `map()` was called on.
- Example 3: Using `map()` to Transform Arrays

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(function (num) {
  return num * 2;
});

console.log(doubled);  // Output: [2, 4, 6, 8]
```

-
- In this example, `map()` creates a new array by doubling each number in the `numbers` array.
  Key Characteristics of `map()`:
  Returns a new array with transformed elements.
  The length of the array returned by `map()` will always be the same as the original array.
  Does not mutate (change) the original array.
-

---

3. Using `forEach()`: Iterating Over Arrays

The `forEach()` method executes a provided function once for each array element. Unlike `map()`, `forEach()` does not return a new array. It is typically used for iterating through arrays when you need to perform side effects, like logging or updating DOM elements.

Syntax:

```
array.forEach(callback);
```

-

`callback`: A function that is executed for each element in the array. It takes three arguments:

`currentValue`: The current element being processed.
`index` (optional): The index of the current element.
`array` (optional): The array `forEach()` was called on.

- Example 4: Using `forEach()` to Iterate Over an Array

```
const fruits = ["apple", "banana", "cherry"];

fruits.forEach(function (fruit) {
  console.log(fruit);
});

// Output:
// apple
// banana
// cherry
```

-
- In this example, `forEach()` simply logs each element in the `fruits` array.
  Key Characteristics of `forEach()`:
  Does not return a new array. It returns `undefined`.
  Typically used for side effects, like logging or DOM manipulation.
  Does not mutate the original array but can manipulate external data.

- ---

4. Difference Between `map()` and `forEach()`

Feature	`map()`	`forEach()`
Return Value	Returns a new array with transformed elements	Returns undefined
Purpose	Used for transformation of array elements	Used for iteration or performing side effects
Chainable	Yes, because it returns a new array	No, it does not return anything
Mutates Original?	No, the original array is not modified	No, but side effects can mutate external values

Typical Use Case	Create a new array with transformed elements	Perform side effects (logging, updating UI)

- Example 5: `map()` vs `forEach()`

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

// Using map() to transform the array
const doubled = numbers.map(function (num) {
  return num * 2;
});
console.log(doubled);  // Output: [2, 4, 6, 8]

// Using forEach() for side effects
numbers.forEach(function (num) {
  console.log(num * 2);  // Logs 2, 4, 6, 8 to the console
});
```

- In the `map()` example, we get a new array (`doubled`).
  In the `forEach()` example, there is no new array; it only logs the doubled values to the console without changing or returning anything.

---

5. Practice Exercises

Exercise 1: Double the Elements

Use `map()` to create a new array where each element in the original array `[1, 2, 3, 4, 5]` is multiplied by 2.

Exercise 2: Log Array Elements

Use `forEach()` to iterate over the array `["a", "b", "c"]` and log each element to the console.

Exercise 3: Convert Temperatures

Given an array of Celsius temperatures, use `map()` to convert them to Fahrenheit using the formula: `F = C * 9/5 + 32`.

Exercise 4: Compare `map()` and `forEach()`

Given the array `[10, 20, 30]`, use both `map()` and `forEach()` to print the values multiplied by 3. Observe how the behavior differs between the two.

---

6. Summary

Higher-Order Functions (HOF) are functions that take other functions as arguments or return them. They provide a powerful tool for abstraction and modularity in JavaScript.

The `map()` method transforms arrays by applying a function to each element and returning a new array. It is useful when you need to modify or transform the data without changing the original array.

The `forEach()` method is used to iterate over arrays and perform side effects, such as logging or manipulating external data. It does not return a new array and is primarily used when transformations are not needed.

Differences between `map()` and `forEach()`:

`map()` returns a new array and is chainable.

`forEach()` returns `undefined` and is used for side effects rather than transformations.

- ---

7. Additional Resources

MDN Web Docs: Higher-Order Functions

MDN Web Docs: `map()`

MDN Web Docs: `forEach()`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

- ---

Advanced Array Methods in JavaScript: `filter()` and `reduce()`

1. Using `filter()`: Creating Subsets of Arrays

The `filter()` method creates a new array with all elements that pass the test provided by a function. It is typically used when you need to create a subset of an array based on certain conditions.

Syntax:

```
array.filter(callback);
```

- `callback`: A function that is called on each element of the array. It takes three arguments:
  - `currentValue`: The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array `filter()` was called on.
- The function should return `true` for elements you want to keep and `false` for those you want to remove.

Example 1: Using `filter()` to Create Subsets

```
const numbers = [1, 2, 3, 4, 5, 6];

// Filter out numbers greater than 3
const filteredNumbers = numbers.filter(function (num) {
  return num > 3;
});

console.log(filteredNumbers);  // Output: [4, 5, 6]
```

-

- In this example, the `filter()` method creates a new array containing numbers greater than 3, without modifying the original `numbers` array.
  Key Characteristics of `filter()`:
    Returns a new array with elements that pass the test.
    The original array remains unchanged.
    Useful for filtering data based on conditions.
- ---

  2. Using `reduce()`: Accumulating Values

  The `reduce()` method executes a reducer function on each element of the array, resulting in a single output value. It's typically used for operations like summing up values or combining data into a single object.

  Syntax:

```
array.reduce(callback, initialValue);
```

- `callback`: A function that executes on each element of the array. It takes four arguments:
  - `accumulator`: The accumulated value from previous iterations.
  - `currentValue`: The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array `reduce()` was called on.

  `initialValue`: The initial value of the accumulator. If not provided, the first element of the array will be used as the initial value.
- Example 2: Using `reduce()` to Sum an Array

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

// Sum all the numbers in the array
const sum = numbers.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);

console.log(sum);  // Output: 15
```

- Example 3: Using `reduce()` to Flatten an Array

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

// Flatten the nested array
const flatArray = nestedArray.reduce(function (accumulator, currentValue) {
  return accumulator.concat(currentValue);
}, []);

console.log(flatArray);  // Output: [1, 2, 3, 4, 5]
```

- In this example, `reduce()` is used to flatten a nested array into a single-level array.
  Key Characteristics of `reduce()`:
    Returns a single value based on the accumulator.
    Can be used for aggregation (sum, product, average) or combining data.
    Requires an initial value for the accumulator to avoid errors.
- ---

3. Difference Table: `map()`, `forEach()`, `filter()`, and `reduce()`
The following table highlights the key differences between `map()`, `forEach()`, `filter()`, and `reduce()`:

Feature	**`map()`**	**`forEach()`**	**`filter()`**	**`reduce()`**
Return Value	Returns a new array with transformed elements	Returns undefined (for side effects)	Returns a new array with filtered elements	Returns a single value (aggregated result)
Purpose	Transforms each element in the array	Iterates over each element for side effects	Filters elements based on a condition	Reduces elements to a single value
Modifies Origin	No	No	No	No

al ?				
Chainable?	Yes	No	Yes	No (but can be combined with other methods)
Use Case	Apply a function to each element and return a new array	Perform actions (like logging) on each element	Create a subset of an array based on a condition	Aggregate values (sum, product, combine data)

- Summary of the Methods:
  - `map()`: Transforms each element and returns a new array.
  - `forEach()`: Iterates over elements to perform side effects (like logging).
  - `filter()`: Returns a new array containing elements that pass a test.
  - `reduce()`: Accumulates a single result from an array (like summing or flattening).
- ---

  4. Practice Exercises

  Exercise 1: Filter Even Numbers

  Write a function that uses `filter()` to create a new array containing only the even numbers from the original array `[1, 2, 3, 4, 5, 6, 7, 8]`.

  Exercise 2: Count Occurrences

  Use `reduce()` to count the occurrences of each item in an array of strings: `["apple", "banana", "apple", "orange", "banana", "banana"]`.

  Exercise 3: Double and Filter

  Given an array `[1, 2, 3, 4, 5]`, first use `map()` to double the values, then use `filter()` to keep only the values greater than 5.

  Exercise 4: Sum of an Array

  Use `reduce()` to find the sum of all numbers in an array `[10, 20, 30, 40]`.

  ---

  5. Summary

`filter()` creates a subset of an array by keeping elements that pass a given test.
`reduce()` accumulates array elements into a single value, useful for aggregation operations like summing or concatenating.
The key differences between `map()`, `forEach()`, `filter()`, and `reduce()`:

`map()` returns a new array of transformed elements.
`forEach()` iterates over elements for side effects but does not return anything.
`filter()` returns a new array with elements that satisfy a condition.
`reduce()` returns a single value, often used for summing, flattening, or aggregating data.

- ---

6. Additional Resources

MDN Web Docs: `filter()`
MDN Web Docs: `reduce()`
MDN Web Docs: `map()`
MDN Web Docs: `forEach()`

- ---

JavaScript Array `sort()` Method

1. Introduction to `sort()`

The `sort()` method sorts the elements of an array in place (i.e., it modifies the original array) and returns the sorted array. The default sort order is based on the Unicode code points of string elements, meaning that `sort()` converts elements to strings and compares their sequences of UTF-16 code units by default.

Syntax:

```
array.sort([compareFunction]);
```

-

`compareFunction` (optional): A function that defines the sort order. If provided, it should return:

A negative value if the first argument should come before the second.
A positive value if the first argument should come after the second.
Zero if the two arguments are considered equal.

- Example 1: Sorting an Array of Strings (Default Behavior)

```
const fruits = ["banana", "apple", "cherry", "date"];
fruits.sort();
console.log(fruits);  // Output: ["apple", "banana", "cherry", "date"]
```

-

- In this example, the strings are sorted in lexicographical (alphabetical) order based on their Unicode values.

---

2. Sorting Numbers

Problem with Default Sorting for Numbers

The default behavior of `sort()` works well for strings but causes problems when sorting numbers, because it sorts numbers as if they were strings.

Example 2: Incorrect Number Sorting

```
const numbers = [25, 1, 100, 50];
numbers.sort();
console.log(numbers);  // Output: [1, 100, 25, 50]
```

- In this example, the numbers are sorted as strings. Therefore, `100` comes before `25` because `"1"` comes before `"2"` in Unicode.

Correct Sorting with a Custom Compare Function

To properly sort numbers, you need to provide a custom compare function that defines how numbers should be compared.

Example 3: Sorting Numbers in Ascending Order

```
const numbers = [25, 1, 100, 50];
numbers.sort(function (a, b) {
  return a - b;
});
console.log(numbers);  // Output: [1, 25, 50, 100]
```

- In this example, the compare function sorts numbers in ascending order by subtracting one number from another (`a - b`).

Example 4: Sorting Numbers in Descending Order

```
const numbers = [25, 1, 100, 50];
numbers.sort(function (a, b) {
  return b - a;
});
console.log(numbers);  // Output: [100, 50, 25, 1]
```

- In this case, the compare function sorts numbers in descending order by subtracting `b` from `a`.

---

3. Sorting Strings

The default behavior of `sort()` works well for strings, but you can provide a custom compare function if you need to control case sensitivity or locale-specific sorting.

Example 5: Case-Insensitive Sorting

```
const fruits = ["Banana", "apple", "cherry", "Date"];
fruits.sort(function (a, b) {
```

```
  return a.toLowerCase().localeCompare(b.toLowerCase());
});
console.log(fruits);  // Output: ["apple", "Banana", "cherry", "Date"]
```

- In this example, the `toLowerCase()` method ensures that the sort is case-insensitive, and `localeCompare()` handles the comparison.

Example 6: Locale-Aware Sorting

```
const fruits = ["banana", "apple", "cherry", "æpple"];
fruits.sort(function (a, b) {
  return a.localeCompare(b);
});
console.log(fruits);  // Output: ["æpple", "apple", "banana", "cherry"]
```

- In this example, `localeCompare()` performs a locale-aware comparison. In some languages, characters like "æ" are treated differently, and `localeCompare()` ensures the sort respects those differences.

---

4. Sorting Objects

When sorting an array of objects, you typically need to sort based on the value of a specific property of the objects. This is done by providing a custom compare function that compares the object properties.

Example 7: Sorting Objects by a Numeric Property

```
const people = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 20 }
];

people.sort(function (a, b) {
  return a.age - b.age;
});

console.log(people);
/* Output:
[
  { name: "Charlie", age: 20 },
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 }
]
*/
```

- In this example, the array of `people` is sorted in ascending order based on the `age` property.

Example 8: Sorting Objects by a String Property

```
const people = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 20 }
];

people.sort(function (a, b) {
  return a.name.localeCompare(b.name);
});

console.log(people);
/* Output:
[
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 20 }
]
*/
```

-
- In this example, the array is sorted in alphabetical order based on the `name` property of each object.

---

5. Common Pitfalls and Considerations

5.1. In-Place Sorting

The `sort()` method modifies the original array in place. If you need to retain the original array, you should make a copy before sorting.

- Example 9: Avoiding In-Place Sorting

```
const numbers = [3, 1, 4, 2];
const sortedNumbers = [...numbers].sort((a, b) => a - b);  // Using the spread operator to copy the array
console.log(sortedNumbers);  // Output: [1, 2, 3, 4]
console.log(numbers);  // Output: [3, 1, 4, 2] (Unchanged)
```

-
- 5.2. Stable vs. Unstable Sort

  JavaScript's `sort()` is not guaranteed to be stable in all implementations (i.e., if two elements are equal, their order might not be preserved).
  However, most modern browsers now implement a stable sort.
- 5.3. Sorting Large Arrays

  Sorting can be computationally expensive, especially for large arrays. Be mindful of performance when dealing with large datasets.
-

---

6. Practice Exercises

Exercise 1: Sorting Numbers in Reverse

Given an array of numbers, write a function to sort the array in descending order.

Exercise 2: Sorting Strings

Given an array of strings, write a function to sort them case-insensitively.

Exercise 3: Sorting Objects by Multiple Properties

Given an array of objects with `name` and `age` properties, write a function to first sort by `age` and then by `name` in case of ties.

---

7. Summary

The `sort()` method sorts an array in place and returns the sorted array.
By default, `sort()` converts elements to strings and sorts them lexicographically, which may cause unexpected results when sorting numbers.
Providing a custom compare function allows for sorting arrays numerically, case-insensitively, or based on object properties.
Be cautious of the in-place modification behavior and consider copying arrays when necessary.

- ---

8. Additional Resources

MDN Web Docs: `sort()`

- ---

Method Chaining with Higher-Order Functions (HOFs) in JavaScript

1. What is Method Chaining?

Definition:

Method chaining is a technique where multiple methods are called sequentially, each performing an action on the result of the previous method. It allows for clean, readable, and concise code.

Why Use Method Chaining?

Improves readability: Method chaining helps to avoid temporary variables and makes code more compact.
Fluent interface: It creates a logical flow of operations, especially when transforming or filtering data.
Reduces code length: Many operations can be combined into a single chain rather than broken down into separate steps.

- Example: Without Method Chaining

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

const doubled = numbers.map(function (num) {
  return num * 2;
});

const filtered = doubled.filter(function (num) {
  return num > 5;
});

console.log(filtered);  // Output: [6, 8, 10]
```

-
- Example: With Method Chaining

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

const result = numbers
```

```
  .map(num => num * 2)
  .filter(num => num > 5);

console.log(result);  // Output: [6, 8, 10]
```

-
- In the chained example, the `map()` and `filter()` methods are combined into a single expression, resulting in cleaner and more compact code.

---

2. Using `map()`, `filter()`, and `reduce()` with Method Chaining

2.1. Chaining `map()` and `filter()`

The `map()` method returns a new array with transformed values, and the `filter()` method returns a new array with values that pass a test. These methods are often chained together when you need to transform data and then filter out certain values.

Example 1: Doubling and Filtering Numbers

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

const result = numbers
  .map(num => num * 2)    // Double each number
  .filter(num => num > 5);  // Keep numbers greater than 5

console.log(result);  // Output: [6, 8, 10]
```

-
- 2.2. Chaining `map()`, `filter()`, and `reduce()`

  The `reduce()` method reduces an array to a single value. When used in a chain, it can help aggregate values after transformation and filtering.

  Example 2: Summing Filtered and Transformed Numbers

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

const sum = numbers
  .map(num => num * 3)          // Triple each number
  .filter(num => num > 10)      // Keep numbers greater than 10
  .reduce((acc, num) => acc + num, 0);  // Sum the remaining numbers

console.log(sum);  // Output: 39
```

-
- Here, the numbers are first tripled, then filtered, and finally summed using `reduce()`.

---

3. Using `forEach()` in Method Chaining

Key Limitation of `forEach()`

Unlike `map()`, `filter()`, and `reduce()`, the `forEach()` method does not return a new array or any value. It simply executes a function for each element. This means that `forEach()` cannot

be chained in the same way as the other methods, because it returns `undefined`.

Proper Use of `forEach()` in Chaining

Though `forEach()` cannot return a new array, it can be used at the end of a chain for performing side effects, such as logging values or updating the DOM.

Example 3: Using `forEach()` After a Chain

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

numbers
  .map(num => num * 2)
  .filter(num => num > 5)
  .forEach(num => console.log(num));

// Output:
// 6
// 8
// 10
```

-
- In this example, `forEach()` is used at the end of the chain to log the transformed and filtered values to the console.

---

4. Combining Multiple Methods with `map()`, `filter()`, and `reduce()`

You can chain multiple methods together for complex data transformations, such as combining `map()`, `filter()`, and `reduce()` to manipulate and summarize data.

Example 4: Product of Filtered Numbers

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8];

// Double the numbers, keep those greater than 10, then calculate their product
const product = numbers
  .map(num => num * 2)              // Double each number
  .filter(num => num > 10)          // Keep numbers greater than 10
  .reduce((acc, num) => acc * num, 1);  // Multiply the remaining numbers

console.log(product);  // Output: 672 (12 * 14 * 16)
```

-
- In this example:
  - `map()` doubles each number.
  - `filter()` keeps numbers greater than 10.
  - `reduce()` multiplies the remaining numbers to get the product.
-

---

5. Best Practices for Method Chaining

Use chaining when it improves readability: Long chains may become difficult to read and maintain. Consider breaking them into smaller, logical parts if they get too complex.

Avoid chaining `forEach()`: Since `forEach()` returns `undefined`, it should not be used in the middle of a chain. Use it only when you need side effects like logging or DOM updates, and place it at the end.
Keep your chains efficient: Avoid unnecessary operations within chains, as they may affect performance.

- ---

6. Differences Between `map()`, `forEach()`, `filter()`, and `reduce()` in Method Chaining

Method	Purpose	Return Value	Can it be Chained?
`map()`	Transforms each element of an array	A new array of transformed elements	Yes
`filter()`	Filters elements based on a condition	A new array of filtered elements	Yes
`reduce()`	Reduces all elements into a single value	A single accumulated value	Yes, but not chainable after
`forEach()`	Executes a function for each array element	`undefined`	No (but can be used at the end)

- ---

7. Practice Exercises
Exercise 1: Chain `map()` and `filter()`
Use `map()` to triple the numbers in an array and then use `filter()` to keep only the numbers that are divisible by 3.
Exercise 2: Chain `map()`, `filter()`, and `reduce()`
Write a chain of methods to first double the numbers, then filter out numbers less than 10, and finally calculate the sum of the remaining numbers.
Exercise 3: Use `forEach()` for Side Effects
Create a chain of `map()` and `filter()` to process an array of strings and then use `forEach()` at the end to log each result.

---

8. Summary

Method chaining allows you to combine multiple method calls in a sequence, passing the result of one method directly into the next.
`map()`, `filter()`, and `reduce()` are key higher-order functions that return new arrays or values, making them suitable for method chaining.
`forEach()` is useful for side effects but does not return a value, so it should not be used in the middle of a chain.
Method chaining improves readability, efficiency, and conciseness, especially when performing multiple transformations on arrays.

- ---

9. Additional Resources
    MDN Web Docs: `map()`
    MDN Web Docs: `filter()`
    MDN Web Docs: `reduce()`

- MDN Web Docs: `forEach()`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach