Day 4

THIS keyword

What is `this`?

- `this` is a keyword in JavaScript that refers to the current object.
- How it's interpreted depends on how a function is called.

Rules for `this`:

1. **Method in an Object:** When a function is a method within an object (`this` refers to the object itself).
2. **Regular Function:** When a function is not part of an object (`this` refers to the global object, which is `window` in browsers and `global` in Node.js).
3. **Constructor Function with `new`:** When a function is called using the `new` operator (used for creating objects), `this` refers to a newly created empty object.

Example:

JavaScript

```
const video = {
  title: "Awesome Video",
  play() {
    console.log(this); // this refers to the video object
  }
};

video.play(); // Outputs the video object
```

String Literals vs. Template Literals:

- **String Literals:**
  - Defined with single or double quotes.
  - Require backslashes (`\n`) for newlines.
  - Need escaping for special characters within quotes (e.g., `\'`).
  - Not ideal for multi-line strings or complex formatting.
- **Template Literals:**
  - Defined with backtick characters ().
  - Allow for newlines directly within the string.

- No need to escape quotes within the template literal.
- Support for embedding expressions using `${expression}`.

Benefits of Template Literals:

- **Improved Readability:** Code closely resembles the formatted output.
- **Easier Multi-Line Strings:** No need for concatenation or extra characters.
- **Simpler String Formatting:** No escaping required for quotes.
- **Dynamic Content:** Can embed expressions (variables, functions) for dynamic values.

Example:

JavaScript

```
// String Literal (Less readable)
const message = "This is my \n first message";

// Template Literal (More readable)
const message = `This is my
first message`;
```

Additional Points:

- Template literals are introduced in ES6 (ECMAScript 2015).
- They are useful for email templates or other formatted string scenarios.

JS array Filter :

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

numbers.filter(function(value){
    return value >= 0; // Filter numbers greater than or equal to 0
});
```

1. `const numbers = [1, -1, 2, 3];`
    - This line declares a constant variable named `numbers` and initializes it with an array containing four numbers: 1, -1, 2, and 3.
2. `numbers.filter(function(value){`
    - This line calls the `filter` method on the `numbers` array. The `filter` method creates a new array with all elements that pass the test implemented by the provided function. Here, it's using an anonymous function as the callback function

for the filtering process. This function will be applied to each element in the `numbers` array.

3. `return value >= 0; // Filter numbers greater than or equal to 0`
    - This line is the body of the anonymous function passed to the `filter` method. It defines the filtering condition. It returns `true` for values greater than or equal to 0 and `false` for values less than 0. If the returned value is `true`, the element will be included in the filtered array; otherwise, it will be excluded.
4. `});`
    - This line closes the anonymous function and the `filter` method call.

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

const filtered = numbers.filter(function(value){
    return value >= 0; // Filter numbers greater than or equal to 0
});
```

JS array map :

The `map()` method in JavaScript is used to iterate over an array and apply a transformation function to each element, returning a new array with the results of applying the function to each element. It doesn't modify the original array; instead, it creates and returns a new array based on the transformation function provided.

Here's a detailed breakdown of how `map()` works:

1. **Syntax**:

   javascriptCopy code

   `const newArray = array.map(callback(currentValue[, index[, array]])[, thisArg])`
    - `array`: The original array you want to iterate over.
    - `callback`: A function that is called for each element in the array. It can take up to three arguments:
        - `currentValue`: The current element being processed in the array.
        - `index` (optional): The index of the current element being processed.
        - `array` (optional): The array `map()` was called upon.
    - `thisArg` (optional): Value to use as `this` when executing `callback`.
2. **Return value**:
    - A new array with each element being the result of the callback function.
3. **Callback function**:
    - The callback function is invoked with three arguments: `currentValue`, `index`, and `array`. However, only `currentValue` is required.

- The callback function is executed for each element in the array.
- It should return the value that will be added to the new array. This value can be of any data type.

4. **Example**:

   javascriptCopy code

   ```
   const numbers = [1, 2, 3, 4, 5]; const doubled = numbers.map(function(num) { return num * 2; }); console.log(doubled); // Output: [2, 4, 6, 8, 10]
   ```

5. **Arrow function syntax**:
   - Arrow functions can be used to provide a more concise syntax, especially for simple callback functions.
   - Example: javascriptCopy code `const doubled = numbers.map(num => num * 2);`
6. **Use cases**:
   - Transforming data: `map()` is commonly used to transform each element in an array into something else, such as doubling each number, converting strings to uppercase, etc.
   - Generating HTML markup: You can use `map()` to generate HTML markup dynamically based on array data.
   - Working with APIs: When working with API data, you might want to transform it into a format that suits your application using `map()`.
7. **Handling sparse arrays**:
   - `map()` skips missing elements in sparse arrays, which can be useful in certain scenarios.
8. **Chaining**:
   - Since `map()` returns a new array, it can be chained with other array methods like `filter()`, `reduce()`, etc., allowing for more complex data transformations in a single statement.
9. **Performance**:
   - `map()` applies the transformation function to each element sequentially. While this is generally fast, performance considerations should be made for large arrays or complex callback functions.
10. **Polyfill**:
    - If you need to support older browsers that don't have `map()` implemented, you can use a polyfill to provide this functionality.

Overall, `map()` is a versatile and powerful method in JavaScript for transforming data in arrays, providing a clean and concise way to manipulate array elements. It's widely used in modern JavaScript development for various data manipulation tasks.

1. **Converting Strings to Uppercase**:

   javascriptCopy code

```
const words = ['hello', 'world', 'javascript']; const uppercaseWords = words.map(word => word.toUpperCase()); console.log(uppercaseWords); // Output: ['HELLO', 'WORLD', 'JAVASCRIPT']
```

2. **Extracting Length of Strings**:

   javascriptCopy code

```
const words = ['apple', 'banana', 'orange']; const wordLengths = words.map(word => word.length); console.log(wordLengths); // Output: [5, 6, 6]
```

3. **Formatting Objects**:

   javascriptCopy code

```
const products = [ { name: 'iPhone', price: 999 }, { name: 'MacBook', price: 1999 }, { name: 'iPad', price: 799 } ]; const formattedProducts = products.map(product => { return { productName: product.name, price: `$${product.price}` }; }); console.log(formattedProducts); /* Output: [ { productName: 'iPhone', price: '$999' }, { productName: 'MacBook', price: '$1999' }, { productName: 'iPad', price: '$799' } ] */
```

4. **Calculating Square Roots**:

   javascriptCopy code

```
const numbers = [1, 4, 9, 16, 25]; const squareRoots = numbers.map(num => Math.sqrt(num)); console.log(squareRoots); // Output: [1, 2, 3, 4, 5]
```

5. **Flattening Arrays**:

   javascriptCopy code

```
const matrix = [[1, 2], [3, 4], [5, 6]]; const flattened = matrix.map(subArray => subArray.join(',')); console.log(flattened); // Output: ['1,2', '3,4', '5,6']
```

6. **Extracting Specific Properties**:

   javascriptCopy code

```
const users = [ { id: 1, name: 'Alice', email: 'alice@example.com' }, { id: 2, name: 'Bob', email: 'bob@example.com' }, { id: 3, name: 'Charlie', email: 'charlie@example.com' } ]; const userIds = users.map(user => user.id); console.log(userIds); // Output: [1, 2, 3]
```

7. **Generating HTML Markup**:

   javascriptCopy code

```
const items = ['apple', 'banana', 'orange']; const listItems = items.map(item => `<li>${item}</li>`); console.log(listItems.join('')); // Output: <li>apple</li><li>banana</li><li>orange</li>
```

JS array reduce

```
const numbers = [ 1, -1 , 2 , 3];
let sum = 0;
for (let n of numbers)
        sum += n;
console.log(sum);
```

1. `const numbers = [1, -1, 2, 3];`
    - This line declares a constant variable named `numbers` and initializes it with an array containing four numbers: 1, -1, 2, and 3.
2. `let sum = ;`
    - This line declares a variable named `sum` using the `let` keyword, indicating that its value can be reassigned. However, there's a syntax error here as there's no initial value assigned to `sum`. It should be initialized to `0` to avoid errors.
3. `for (let n of numbers)`
    - This line initiates a `for...of` loop, which iterates over each element (`n`) in the `numbers` array.
4. `sum += n;`
    - Inside the loop, this line adds each element (`n`) of the `numbers` array to the `sum` variable. The `+=` operator is shorthand for adding the current value of `n` to the current value of `sum` and assigning the result back to `sum`. Essentially, this statement calculates the sum of all elements in the `numbers` array.
5. `console.log(sum);`
    - This line logs the final value of `sum` to the console after the loop completes. It displays the sum of all elements in the `numbers` array.

```
// 1st method
const numbers = [ 1, -1 , 2 , 3];
let sum = 0;
for (let n of numbers)
        sum += n;
console.log(sum);

//2nd method

const totalSum = numbers.reduce((accumulator , currentValue) => {
        return accumulator + currentValue;
}, 12 );  // the 12 here sets the value of accumulator to 12
console.log(totalSum);
```

the accumulator shall be the starting point of the loop . we have set the value to 12 , so in the first loop , there is 12 +1 , in the second loop , the accumulator shall become 13 and the currentValue will be -1 and the operation is repeated