1. Swapping Values Without a Temporary Variable

```
let a = 1, b = 2;

[a, b] = [b, a];

// Output: a = 2, b = 1
```

without needing a temporary variable. It's a neat trick that makes your code cleaner and more concise. The `[a, b] = [b, a]` syntax swaps their values by destructuring the array on the right-hand side and assigning it to the left-hand side.

2. Object Destructuring for Easier Data Access

```
const {name, age} = {name: 'John', age: 30};

// Output: name = 'John', age = 30
```

Here, object destructuring is used to extract `name` and `age` properties from an object directly into variables. This approach simplifies accessing object properties and enhances code readability.

3. Cloning Objects in a Snap

```
const originalObj = {name: 'Jane', age: 22};

const clonedObj = {...originalObj};

// Output: clonedObj = {name: 'Jane', age: 22}
```

The spread operator (`...`) is utilized to create a shallow clone of `originalObj` . This technique copies all enumerable own properties from the original object to a new object.

So can we not simply assign the old object to the new one?

```
const clonedObj = originalObj;
clonedObj.name = "George";

// Output: originalObj = {name: 'George', age: 22}
```

You can do this but `clonedObj` will simply be a reference to `originalObj` so if you tried modifying the `name` property on `clonedObj` you will actually be modifying the `originalObj` !

4. Merging Objects Made Simple

```
const obj1 = {name: 'Jane'};
const obj2 = {age: 22};

const mergedObj = {...obj1, ...obj2};
```

```
// Output: mergedObj = {name: 'Jane', age: 22}
```

Similar to cloning, the spread operator is used to merge `obj1` and `obj2` into a new object. If there are overlapping properties, the properties from the last object will override the previous ones.

5. Cleaning Up Arrays

```
const arr = [0, 1, false, 2, '', 3];

const cleanedArray = arr.filter(Boolean);

// Output: cleanedArray = [1, 2, 3]
```

This clever trick uses `Array.prototype.filter()` with the `Boolean` constructor as a callback. It removes all falsy values (`0`, `false`, `null`, `undefined`, `''`, `NaN`) from an array.

6. Transforming NodeList to Array

```
const nodesArray = [...document.querySelectorAll('div')];
```

The spread operator is used to convert a `NodeList` (returned by `document.querySelectorAll`) into a JavaScript array, enabling array methods like `map`, `filter`, etc.

7. Checking Arrays for Specific Conditions

```
const arr = [1, 2, 3, -1, 4];

const hasNegativeNumbers = arr.some(num => num < 0);

// Output: hasNegativeNumbers = true
```

The `.some()` method checks if at least one element in the array passes the test implemented by the provided function (here, being negative), while `.every()` checks if all elements pass the test (being positive).

```
const arr = [1, 2, 3, -1, 4];

const allPositive = arr.every(num => num > 0);

//Output: allPositive = false
```

8. Copying Text to Clipboard

```
navigator.clipboard.writeText('Text to copy');
```

This line uses the Clipboard API to programmatically copy text to the user's clipboard. It's a modern approach for interacting with the clipboard, making copying text seamless and efficient.

9. Creating a Unique Array

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

const unique = [...new Set(arr)];

// Output: unique = [1, 2, 3, 4, 5]
```

This utilizes the `Set` object, which stores unique values, combined with the spread operator to convert it back to an array. It's an elegant way to remove duplicates from an array.

10. Finding the Intersection of Two Arrays

```
const arr1 = [1, 2, 3, 4];
const arr2 = [2, 4, 6, 8];

const intersection = arr1.filter(value => arr2.includes(value));

// Output: intersection = [2, 4]
```

This example uses `Array.prototype.filter()` to find common elements in `arr1` and `arr2`. The callback function checks if `arr2` includes each element of `arr1`, resulting in an array of intersecting values.

11. Sum of Array Values

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

const sum = arr.reduce((total, value) => total + value, 0);

// Output: sum = 10
```

This one-liner uses the `reduce` method to accumulate all values in an array into a single sum. The `reduce` method takes a callback function with two parameters: the accumulator (`total`) and the current value (`value`). Starting from an initial value of 0, it iterates through the array, adding each element to the total sum.

12. Conditional Object Property

```
const condition = true;
const value = 'Hello World';

const newObject = {...(condition && {key: value})};

// Output: newObject = { key: 'Hello World' }
```

This clever use of the spread operator (`...`) and short-circuit evaluation allows you to conditionally add a property to an object. If `condition` is true, `{key: value}` is spread into the object; otherwise, nothing is added.

13. Dynamic Object Key

```
const dynamicKey = 'name';
const value = 'John Doe';

const obj = {[dynamicKey]: value};

// Output: obj = { name: 'John Doe' }
```

This syntax, known as computed property names, allows you to use a variable as an object key. The square brackets around `dynamicKey` evaluate the expression inside to use it as the property name.

14. Online Status Checker

```
const isOnline = navigator.onLine ? 'Online' : 'Offline';

// Output: isOnline = 'Online' or 'Offline'
```

Using the ternary operator, this piece of code checks the browser's online status with `navigator.onLine`, returning 'Online' if true and 'Offline' otherwise. It's a quick way to dynamically check the user's connection status. There are also event listeners availabe to be used.

15. Confirm Before Page Leave

```
window.onbeforeunload = () => 'Are you sure you want to leave?';
```

This line of code hooks into the window's `onbeforeunload` event, which triggers a confirmation dialog when the user attempts to leave the page, helping to prevent data loss from unsaved changes.

16. Sum of Object Values by Key

```
const arrayOfObjects = [{x: 1}, {x: 2}, {x: 3}];

const sumBy = (arr, key) => arr.reduce((acc, obj) => acc + obj[key], 0);

sumBy(arrayOfObjects, 'x'));
// Output: 6
```

This function uses `reduce` to sum values of a specific key across objects in an array. It's a flexible way to calculate totals from an array of objects based on a given key.

17. Parse Query String to Object

```
const query = 'name=John&age=30';

const parseQuery = query => Object.fromEntries(new URLSearchParams(query));

// Output: parseQuery = { name: 'John', age: '30' }
```

This expression converts a query string into an object. `URLSearchParams` parses the query, and `Object.fromEntries` transforms the iterable pairs into an object, making URL parameter retrieval straightforward.

18. Convert Seconds to Time String

```
const seconds = 3661;

const toTimeString = seconds => new Date(seconds * 1000).toISOString().substr(11

toTimeString(seconds));
// Output: '01:01:01'
```

This line converts seconds into a HH:MM:SS format string. It creates a new Date object at the epoch plus the given seconds, converts it to an ISO string, and extracts the time portion.

19. Maximum Value in Object

```
const scores = {math: 95, science: 99, english: 88};

const maxObjectValue = obj => Math.max(...Object.values(obj));

maxObjectValue(scores));
// Output: 99
```

This one-liner finds the maximum value among an object's values. `Object.values(obj)` extracts the values as an array, which is then spread into `Math.max` to find the highest number.

20. Check if Object Contains Value

```
const person = {name: 'John', age: 30};

const hasValue = (obj, value) => Object.values(obj).includes(value);

hasValue(person, 30);
// Output: true
```

This function checks if a given value exists within the values of an object. `Object.values(obj)` gets an array of the object's values, and `includes(value)` checks if the value is in that array.

21. Boosting Scores Conditionally

```
const scores = [45, 75, 62, 55, 90];

const updatedScores = scores.map(score => score < 60 ? score + 20 : score);

// Output: updatedScores = [65, 75, 62, 75, 90]
```

This one-liner uses the `Array.map()` method to iterate over each score in the `scores` array. The ternary operator (`condition ? exprIfTrue : exprIfFalse`) is used within the callback function to check if a score is below 60. If true, it adds 20 to the score; otherwise, it returns the score unchanged. This is a great way to apply conditional logic inline during array transformations. Perfect for adjusting data based on conditional logic without needing a verbose `if-else` block.

22. Safely Accessing Nested Object Properties

```
const user = { profile: { name: 'John Doe' } };

const userName = user.profile?.name ?? 'Anonymous';

// Output: userName = 'John Doe'
```

This one-liner demonstrates the use of the optional chaining operator (`?.`) to safely access `name` within `user.profile`. If `user.profile` is `undefined` or `null`, it short-circuits and returns `undefined`, avoiding a potential `TypeError`.

The nullish coalescing operator (`??`) then checks if the left-hand side is `null` or `undefined`, and if so, defaults to 'Anonymous'. This ensures a fallback value without falsely triggering on other falsy values like `''` or `0`. This is ideal for accessing deeply nested properties in data structures where some intermediate properties may not exist.

In JavaScript, both the nullish coalescing operator (`??`) and the logical OR (`||`) can be used to provide default values, but they differ in how they handle falsy values.

In the above example, changing `??` to `||` would change the behaviour slightly. The logical OR (`||`) operator returns the right-hand operand if the left-hand operand is falsy. Falsy values in JavaScript include `null`, `undefined`, `0`, `NaN`, `''` (empty string), and `false`. This means that the `||` operator will return the right-hand operand for any of these falsy values, not just `null` or `undefined`.

23. Conditional Execution

```
const isEligible = true;

isEligible && performAction();

// performAction is called if isEligible is true
```

Leveraging the logical AND (`&&`) operator, this pattern executes `performAction()` only if `isEligible` evaluates to `true` . It's a concise way to conditionally execute a function without an `if` statement. This could be useful for executing functions based on conditions, especially in event handlers or callbacks.

24. Generating a Range of Numbers

```
const range = Array.from({ length: 5 }, (_, i) => i + 1);

// Output: range = [1, 2, 3, 4, 5]
```

`Array.from()` creates a new array from an array-like or iterable object. Here, it takes an object with a `length` property and a mapping function. The mapping function (`(_, i) => i + 1`) uses the index (`i`) to generate numbers from 1 to 5. The underscore (`_`) is a convention indicating that the parameter is not used.

25. Implementing Promises with Timeout

```
const timeout = (promise, ms) => Promise.race([
  promise,
  new Promise((_, reject) => setTimeout(() => reject(new Error("Timeout")), ms))
]);

timeout(fetch('https://api.example.com'), 5000).then(handleResponse).catch(handl
```

This piece of code creates a promise that rejects with a "Timeout" error if not resolved within a specified number of milliseconds. It uses `Promise.race()` to race the given `promise` against a timeout promise that automatically rejects after `ms` milliseconds. This is perfect for adding timeout behaviour to fetch requests or any asynchronous operation that may hang or take too long.

If you've not used Promises before (in JavaScript) then the above might need a bit more explanation.

A Promise in JavaScript represents the eventual completion (or failure) of an asynchronous operation and its resulting value. A Promise can be in one of these states:

- **Pending**: The initial state of a Promise. The operation has not completed yet.
- **Fulfilled**: The operation completed successfully, and the Promise has a value.
- **Rejected**: The operation failed, and the Promise has a reason for the failure.

Promises allow you to attach callbacks to handle the fulfilled value or the reason for rejection, without having to directly handle the asynchronous nature of the operation.

Using `Promise.race()`

The `Promise.race()` method takes an iterable of Promise objects and returns a new Promise that resolves or rejects **as soon as one of the Promises in the iterable resolves or rejects**, with the value or reason from that Promise.

How the `timeout` Function Works

The `timeout` function provided in the example wraps any Promise (in this case, a `fetch` request) with a timeout functionality. It creates a race condition between the provided Promise and a new Promise that automatically rejects after a specified timeout period (in milliseconds).

This approach is particularly useful for ensuring that your application can gracefully handle situations where an asynchronous operation might hang or take an excessively long time, by providing a mechanism to "give up" and handle the timeout as an error.

26. Extracting File Extension

```
const fileName = 'example.png';

const getFileExtension = str => str.slice(((str.lastIndexOf(".") - 1) >>> 0) + 2

// Output: getFileExtension = 'png'
```

This expression extracts the file extension from a string. It finds the last occurrence of the dot character (.) and slices the string from that position to the end. The bitwise operator (>>>) ensures the operation is safe even if the dot is not found, effectively returning an empty string in such cases.

27. Checking if the Current Tab is Focused

```
const isTabFocused = () => document.hasFocus();

// Output: true (if the tab is focused), false otherwise
```

Utilizes the `document.hasFocus()` method to check if the document (or tab) has focus. This is a straightforward way to determine if the user is actively viewing or interacting with the page. Can be used to pause or resume activities based on user presence, like stopping video playback when the user switches tabs.

28. Toggling a Class Name on an Element

```
const element = document.querySelector('.my-element');

const toggleClass = (el, className) => el.classList.toggle(className);

toggleClass(element, 'active');
```

This function uses the `classList.toggle()` method to add or remove a class from an element's class list. If the class exists, it's removed; if it doesn't, it's added. This is a neat way to handle dynamic class changes based on user interactions or application state. Perfect for implementing responsive design elements, such as menus or modals, that show or hide based on user actions.

And that’s it!

Twenty-eight JavaScript one-liners that deliver impactful functionality in a short and efficient way. Hopefully, you learned something today!

Javascript Tips Javascript Development Typescript Front End Development Frontend

Written by Mate Marschalko

Follow

1.4K Followers

Senior Creative Developer, Generative AI, Electronics with over 15 years experience | JavaScript, HTML, CSS