Useful JavaScript Snippets for Web Developers

A curated collection of essential vanilla JavaScript code snippets, organized into categories for easy reference. Each snippet includes a practical example and a detailed explanation.

📂 1. DOM Manipulation

Working with the Document Object Model is a cornerstone of front-end development.

1.1. Select a Single Element

Select the first element that matches a CSS selector.

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

What it does: Finds and returns the first DOM element that matches the specified CSS selector (e.g., #id, .class, tag). If no match is found, it returns null.

1.2. Select Multiple Elements

Select all elements that match a CSS selector.

```
const elements = document.querySelectorAll('.item');
```

What it does: Returns a static NodeList representing a list of the document's elements that match the specified group of selectors. You can iterate over this list with forEach.

1.3. Add or Remove a CSS Class

Toggle, add, or remove a class from an element's classList.

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

// Add a class
element.classList.add('active');

// Remove a class
element.classList.remove('active');

// Toggle a class
element.classList.toggle('highlight');
```

```
// Check if a class exists
if (element.classList.contains('active')) {
  console.log('Element is active!');
}
```

What it does: The classList property provides methods to easily manage an element's CSS classes without manually manipulating the className string.

1.4. Create and Append a New Element

Create a new DOM element and add it to another element.

```
const newDiv = document.createElement('div');
newDiv.textContent = 'Hello, World!';
newDiv.classList.add('new-item');

const container = document.querySelector('.container');
container.appendChild(newDiv);
```

What it does: createElement() creates a new element node. appendChild() adds this new node to the end of the list of children of a specified parent node.

1.5. Get or Set Text Content

Change the text inside an element.

```
const heading = document.querySelector('h1');

// Get text content
const currentText = heading.textContent; // "Original Title"

// Set new text content
heading.textContent = 'New Awesome Title';
```

What it does: textContent gets the text content of an element and all its descendants, or sets the text content of an element, replacing any existing children.

1.6. Get or Set HTML Content

Change the HTML inside an element.

```
const contentArea = document.querySelector('#content');

// Get HTML content
const currentHTML = contentArea.innerHTML;

// Set new HTML content
contentArea.innerHTML = '<h2>New Section</h2><p>This is some
<strong>bold</strong> text.</p>';
```

What it does: innerHTML gets or sets the HTML syntax describing the element's descendants. Be cautious when setting innerHTML with user-provided data to avoid XSS vulnerabilities.

1.7. Get, Set, or Remove a Data Attribute

Work with custom data-* attributes.

```
const userCard = document.querySelector('#user-card');

// Set a data attribute
userCard.dataset.userId = '12345';

// Get a data attribute
const userId = userCard.dataset.userId; // "12345"

// Remove a data attribute
delete userCard.dataset.userId;
```

What it does: The dataset property provides a simple way to access all custom data attributes (data-*) set on an element. It automatically converts kebab-case (data-user-id) to camelCase (userId).

🖱️ 2. Event Handling

Making web pages interactive.

2.1. Add an Event Listener

Execute a function when an event occurs (e.g., a click).

```
const myButton = document.querySelector('#myButton');

myButton.addEventListener('click', () => {
  console.log('Button was clicked!');
});
```

What it does: Attaches an event handler to the specified element. The handler function is executed whenever the specified event is delivered to the target.

2.2. Event Delegation

Listen for events on a parent element instead of multiple child elements. This is more efficient and handles dynamically added elements.

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

list.addEventListener('click', (event) => {
  // Check if the clicked element is an 'LI'
  if (event.target && event.target.matches('li.item')) {
    console.log('List item clicked:', event.target.textContent);
  }
});
```

What it does: A single event listener is placed on a parent element. When an event is triggered on a child, it "bubbles up" to the parent. We can then check event.target to identify which child was the source of the event.

2.3. Prevent Default Action

Stop the browser's default behavior for an event.

```
const form = document.querySelector('form');

form.addEventListener('submit', (event) => {
  // Prevent the form from submitting and reloading the page
  event.preventDefault();
  console.log('Form submission handled by JavaScript.');
});
```

What it does: event.preventDefault() tells the browser not to execute the default action associated with the event (e.g., following a link on click, submitting a form).

2.4. Get Mouse Position

Find the X and Y coordinates of the mouse pointer.

```
document.addEventListener('mousemove', (event) => {
  const x = event.clientX;
  const y = event.clientY;
  // console.log(`Mouse position: ${x}, ${y}`);
});
```

What it does: The mousemove event fires continuously as the mouse moves over an element. The event object contains clientX and clientY properties for the viewport coordinates.

🗃️ 3. Array Methods

Powerful built-in methods for working with arrays.

3.1. Loop Over an Array: forEach

Execute a function once for each array element.

```
const fruits = ['apple', 'banana', 'cherry'];
fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);
});
```

What it does: A simple and readable way to iterate over all items in an array. It does not return a new array.

3.2. Transform an Array: map

Create a new array by performing an operation on each element of the original array.

```
const numbers = [1, 4, 9, 16];
const roots = numbers.map(num => Math.sqrt(num));
// roots is now [1, 2, 3, 4]
```

What it does: map() creates a new array populated with the results of calling a provided function on every element in the calling array.

3.3. Filter an Array: filter

Create a new array with all elements that pass a test.

```
const ages = [32, 33, 16, 40];
const adults = ages.filter(age => age >= 18);
// adults is now [32, 33, 40]
```

What it does: filter() creates a new array with all elements that return true from the provided callback function.

3.4. Find an Element: find

Return the first element in an array that satisfies a test.

```
const products = [
  { id: 'a', name: 'Book' },
  { id: 'b', name: 'Pen' },
  { id: 'c', name: 'Paper' }
];
const foundProduct = products.find(product => product.id === 'b');
// foundProduct is { id: 'b', name: 'Pen' }
```

What it does: Returns the value of the first element that passes the test. Otherwise undefined is returned.

3.5. Reduce an Array to a Single Value: reduce

Execute a reducer function on each element of the array, resulting in a single output value.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) => accumulator +
currentValue, 0);
// sum is 10
```

What it does: Great for calculating totals, averages, or transforming an array into a single object. The accumulator holds the return value of the previous iteration.

3.6. Check if Any Element Passes a Test: some

Tests whether at least one element in the array passes the test.

```
const numbers = [1, 2, 3, 4, 5];
const hasEvenNumber = numbers.some(num => num % 2 === 0);
// hasEvenNumber is true
```

What it does: Returns true if the callback function returns a truthy value for at least one element in the array. Otherwise, it returns false.

3.7. Check if All Elements Pass a Test: every

Tests whether all elements in the array pass the test.

```
const numbers = [2, 4, 6, 8];
const allAreEven = numbers.every(num => num % 2 === 0);
// allAreEven is true
```

What it does: Returns true if the callback function returns a truthy value for every array element. Otherwise, it returns false.

3.8. Create a Flat Array: flat

Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

```
const nestedArray = [1, 2, [3, 4, [5, 6]]];
const flatArray = nestedArray.flat(2);
// flatArray is [1, 2, 3, 4, 5, 6]
```

What it does: Simplifies nested arrays into a single-level array. flat(Infinity) will flatten all levels.

📝 4. String Manipulation

Common tasks for working with text.

4.1. Check if a String Contains a Substring

```
const sentence = 'The quick brown fox jumps over the lazy dog.';
const hasFox = sentence.includes('fox'); // true
```

What it does: The includes() method determines whether one string may be found within another string, returning true or false as appropriate. It's case-sensitive.

4.2. Get a Substring

```
const str = 'Hello, world!';
const world = str.substring(7, 12); // "world"
const sliceWorld = str.slice(7, 12); // "world"
```

What it does: substring() and slice() both extract a part of a string and return a new string. slice() is slightly more flexible as it can accept negative indexes.

4.3. Split a String into an Array

```
const csv = 'one,two,three,four';
const values = csv.split(',');
// values is ['one', 'two', 'three', 'four']
```

What it does: The split() method divides a String into an ordered list of substrings, puts these substrings into an array, and returns the array.

4.4. Trim Whitespace

```
const greeting = '   Hello world!   ';
const trimmed = greeting.trim(); // "Hello world!"
```

What it does: trim() removes whitespace from both ends of a string. trimStart() and trimEnd() can be used to trim from only one end.

4.5. Pad a String

```
const str = '5';
const paddedStart = str.padStart(4, '0'); // "0005"
```

```
const paddedEnd = str.padEnd(3, '!');   // "5!!"
```

What it does: Pads the current string with another string (multiple times, if needed) until the resulting string reaches the given length.

🔮 5. Asynchronous JavaScript

Handling operations that take time, like network requests.

5.1. Fetch API (GET Request)

Make a simple GET request to an API endpoint.

```
async function getData() {
  try {
    const response = await fetch('https://api.github.com/users/google');
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Could not fetch data:', error);
  }
}

getData();
```

What it does: The fetch() API provides a modern, promise-based interface for making network requests. The async/await syntax makes the asynchronous code look and behave more like synchronous code, which is easier to read and reason about.

5.2. Fetch API (POST Request)

Send data to an API endpoint using a POST request.

```
async function postData(url = '', data = {}) {
  try {
    const response = await fetch(url, {
      method: 'POST',
```

```
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(data)
    });
    return await response.json();
  } catch (error) {
    console.error('Error posting data:', error);
  }
}

postData('https://jsonplaceholder.typicode.com/posts', { title: 'foo', body: 'bar', userId:
1 })
  .then(data => {
    console.log(data);
  });
```

What it does: To send data, you provide a second argument to fetch() with an options object. This includes the method, headers (to specify the data type), and the body (which must be stringified).

🛠️ 6. Utility Functions

Helpful functions for common, everyday tasks.

6.1. Generate a Random Number in a Range

```
function getRandomNumber(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

const diceRoll = getRandomNumber(1, 6);
```

What it does: Math.random() returns a number between 0 (inclusive) and 1 (exclusive). This formula scales that result to fit within the desired min and max range.

6.2. Debounce a Function

Limit the rate at which a function gets called. Useful for events like window resizing or

search input.

```
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}

// Example usage:
window.addEventListener('resize', debounce(() => {
  console.log('Window resized! (but this only logs after 300ms of no resizing)');
}, 300));
```

What it does: Creates a "cooldown" period. The wrapped function will only execute after it has not been called for a specified delay. Each new call resets the timer.

6.3. Copy Text to Clipboard

```
function copyToClipboard(text) {
  // Use the modern Clipboard API if available
  if (navigator.clipboard) {
    navigator.clipboard.writeText(text).then(() => {
      console.log('Text copied to clipboard');
    }).catch(err => {
      console.error('Failed to copy text: ', err);
    });
  } else {
    // Fallback for older browsers
    const textArea = document.createElement('textarea');
    textArea.value = text;
    document.body.appendChild(textArea);
    textArea.select();
    try {
      document.execCommand('copy');
      console.log('Text copied (fallback method)');
```

```
    } catch (err) {
      console.error('Fallback failed: ', err);
    }
    document.body.removeChild(textArea);
  }
}

copyToClipboard('Hello, this is the text to be copied!');
```

What it does: Provides a robust way to copy text. It prioritizes the modern, secure navigator.clipboard API and includes a fallback using a temporary textarea for older browser compatibility.

6.4. Check if an Element is in Viewport

```
function isElementInViewport(el) {
  const rect = el.getBoundingClientRect();
  return (
    rect.top >= 0 &&
    rect.left >= 0 &&
    rect.bottom <= (window.innerHeight || document.documentElement.clientHeight)
&&
    rect.right <= (window.innerWidth || document.documentElement.clientWidth)
  );
}

// Example usage:
const myElement = document.querySelector('#special-section');
if (isElementInViewport(myElement)) {
  console.log('Element is visible!');
}
```

What it does: getBoundingClientRect() returns the size of an element and its position relative to the viewport. This function checks if all four sides of the element are currently within the visible boundaries of the window.