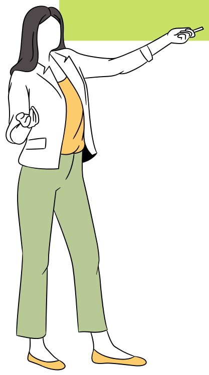
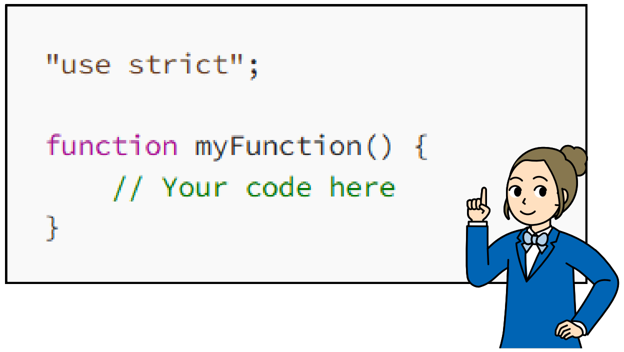
JAVASCRIPT PERFORMANCE TIPS



@DimpleKumari
Forming a network of fantastic coders.

1) Use Strict Mode

Enabling strict mode in JavaScript catches common coding bloopers, prevents the use of undeclared variables, and makes your code run faster.



```
"use strict";

function myFunction() {
    // Your code here
}
```

“use strict;” can tell the browser to execute in strict mode, which can improve the performance

2) Minimize DOM Manipulation

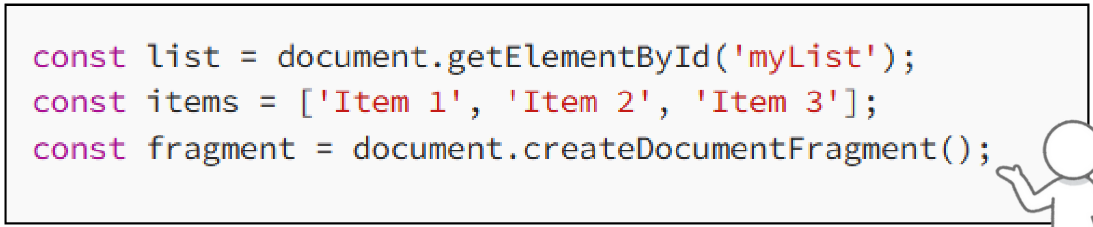
Manipulating the Document Object Model (DOM) is one of the slowest operations in JavaScript. Reducing the number of direct DOM manipulations can significantly improve performance.

Instead of:

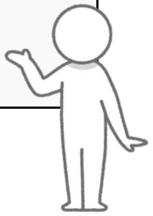
```
const list = document.getElementById('myList');
const items = ['Item 1', 'Item 2', 'Item 3'];

items.forEach(item => {
    const li = document.createElement('li');
    li.textContent = item;
    list.appendChild(li);
});
```

2) Minimize DOM Manipulation

Use Document Fragments



```
const list = document.getElementById('myList');
const items = ['Item 1', 'Item 2', 'Item 3'];
const fragment = document.createDocumentFragment();
```



By using a document fragment, you batch your DOM updates, which is much more efficient.
Personal Note: After switching to document fragments in a dynamic list, I noticed a significant reduction in rendering time, especially with large datasets.

3) Use Event Delegation

Attaching event listeners to multiple DOM elements can be inefficient. Event delegation allows you to handle events at a higher level in the DOM.

Instead of:

```
const buttons = document.querySelectorAll('.myButton');
buttons.forEach(button => {
    button.addEventListener('click', function() {
        // Handle click
    });
});
```

Use Event Delegation:

```
document.body.addEventListener('click', function(event) {
    if (event.target.classList.contains('myButton')) {
        // Handle click
    }
});
```

@DimpleKumari

Forming a network of fantastic coders.

4) Avoid Memory Leaks

Attaching event listeners to multiple DOM elements can be inefficient. Event delegation allows you to handle events at a higher level in the DOM.

Common Pitfall:

```
let element = document.getElementById('myElement');
element.addEventListener('click', function() {
    console.log('Clicked!');
});
// Later in the code
element = null; // This doesn't remove the event listener
```

Proper Cleanup:

```
let element = document.getElementById('myElement');
function handleClick() {
    console.log('Clicked!');
}
element.addEventListener('click', handleClick);
// Later in the code
element.removeEventListener('click', handleClick);
element = null;
```

@DimpleKumari
Forming a network of fantastic coders.

5) Optimize Loops

Loops can be performance bottlenecks. Simple changes can make them more efficient.

Instead of:

```
for (let i = 0; i < array.length; i++) {
    // Do something with array[i]
}
```

Cache the Length:

```
for (let i = 0, len = array.length; i < len; i++) {
    // Do something with array[i]
}
```

@DimpleKumari

Forming a network of fantastic coders.

6) Debounce and Throttle Expensive Functions

For functions that are called frequently, like window resizing or scrolling, use debouncing or throttling to limit how often they run.

Debounce Example:

```
function debounce(func, delay) {
    let timeout;
    return function() {
        clearTimeout(timeout);
        timeout = setTimeout(func, delay);
    }
}

window.addEventListener('resize', debounce(function() {
    // Handle resize
}, 250));
```

@DimpleKumari
Forming a network of fantastic coders.

6) Debounce and Throttle Expensive Functions

For functions that are called frequently, like window resizing or scrolling, use debouncing or throttling to limit how often they run.

Throttle Example:

```
function throttle(func, limit) {
    let inThrottle;
    return function() {
        if (!inThrottle) {
            func();
            inThrottle = true;
            setTimeout(() => inThrottle = false, limit);
        }
    }
}
window.addEventListener('scroll', throttle(function() {
    // Handle scroll
}, 250));
```

7) Use Asynchronous Code Wisely

Non-blocking code keeps your application responsive. Use asynchronous programming features like async/await and Promises.

```
async function fetchData() {
    try {
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        // Process data
    } catch (error) {
        console.error(error);
    }
}
```

By handling operations asynchronously, you prevent blocking the main thread.

@DimpleKumari

Forming a network of fantastic coders.

THANK YOU FOR READING!

If you found this informative and valuable, I'd love for you to connect with me. Follow me Medium, Codepen, and connect with me on LinkedIn to stay updated on the latest in web development, interviews, and more.

Let's connect!

💼 **LinkedIn** — https://www.linkedin.com/in/dimple-kumari/

🔗 **Medium** — https://medium.com/@dimplekumari0228

✍️ **Codepen** — https://codepen.io/DIMPLE2802

@DimpleKumari

Forming a network of fantastic coders.

DIMPLE KUMARI

Forming a network of fantastic coders.