

# Writing Clean Javascript Codes



**A Simple Guide (Part 1)**

# Introduction

"Messy code doesn't just slow down your project, it slows down your progress."

As we all know, JavaScript is one of the world's most popular programming languages, powers everything from engaging web applications to robust server-side solutions.

But with great power comes the responsibility of writing **clean, readable, maintainable** code; a task that can either make your life easier or leave you untangling a web of confusion.

As a developer with JavaScript as my primary language, I've experienced firsthand the transformation that comes from improving my coding skills.

Over time (**and through plenty of trial and error**), I've learned that writing clean code isn't just a best practice; it's the secret ingredient to building scalable, bug-free applications and keeping your sanity intact.

The journey is ongoing, but the results? Worth every effort.

# What does writing clean codes mean?

Clean code is code that is easy to read, understand, and modify. It ensures that your work is not only functional but also intuitive for others (or even your future self) to pick up and enhance.

As projects grow in size and complexity, messy code can become a nightmare to debug and maintain, slowing down development and increasing the risk of introducing new bugs.

The goal of writing clean JavaScript is to simplify collaboration, improve performance, and future-proof your applications. In this guide, we'll explore **10 essential best practices** for writing clean JavaScript code, this is the **Part 1**, watch out for the other part.

Each practice will be explained comprehensively, with real-world examples and actionable tips to help you write code that's efficient, readable, and professional.

# 1. Use Descriptive Variable and Function Names

Naming variables and functions clearly and descriptively is an important aspect of writing clean code.

Giving them meaningful names makes it clear what data is stored or what task is performed.

When names are well-chosen, other developers can quickly understand what the variable or function is doing, and how it is related to the rest of the code.

## Key Rules for Naming:


- Use **nouns** for **variables** representing things, e.g., `userName`, `orderTotal`.
- Use **verbs** for **function** names to represent actions, e.g., `getUserData()`, `sendEmail()`.

For example:



```
1 // Bad Example
2 let n = 1000;
3 function p(x, y) {
4     return x * y;
5 }
6 console.log(p(n, 5));
```

We don't know exactly what "p" or "n" does? "n" is too generic, does it represent "number," "name," or something else?



```
1 // Good Example
2 const employeeSalary = 1000;
3 function calculateAnnualBonus(salary, bonusMultiplier) {
4     return salary * bonusMultiplier;
5 }
6 console.log(calculateAnnualBonus(employeeSalary, 5));
```

Here, **employeeSalary** immediately tells us the variable stores an employee's salary while **calculateAnnualBonus()** explains the function's purpose without needing comments.

## 2. Follow Consistent Formatting

onsistent formatting in JavaScript code ensures that your code is easier to read, debug, and maintain for yourself and others.

It's about adopting a clear and uniform structure throughout your codebase so that it looks organized and predictable.

### Why Follow Consistent Formatting?

- **Improves Readability:** Clean and consistently formatted code is easier to read and understand, especially for larger teams or when revisiting code after a long time.
- **Minimizes Errors:** Proper formatting reduces the chances of syntax errors or logical mistakes that arise from messy or disorganized code.
- **Speeds Up Debugging:** Well-formatted code makes it easier to locate issues and understand the flow.



- **Supports Team Collaboration:** Consistent formatting creates a shared coding standard, reducing misunderstandings among developers.
- **Enhances Scalability:** As your project grows, consistently formatted code becomes easier to maintain and extend.

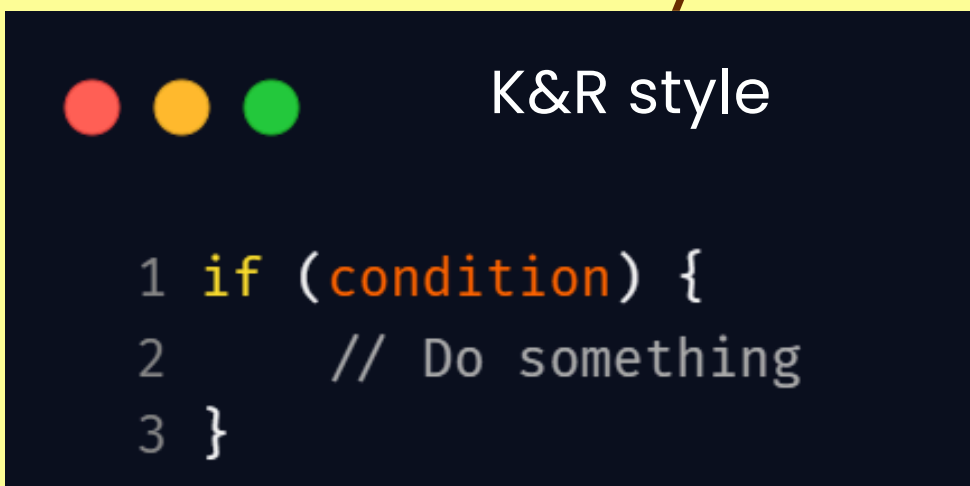
## Tips for Maintaining Consistent Formatting:

- **Indentation:** Use a consistent number of spaces (usually 2 or 4) or tabs for code blocks.



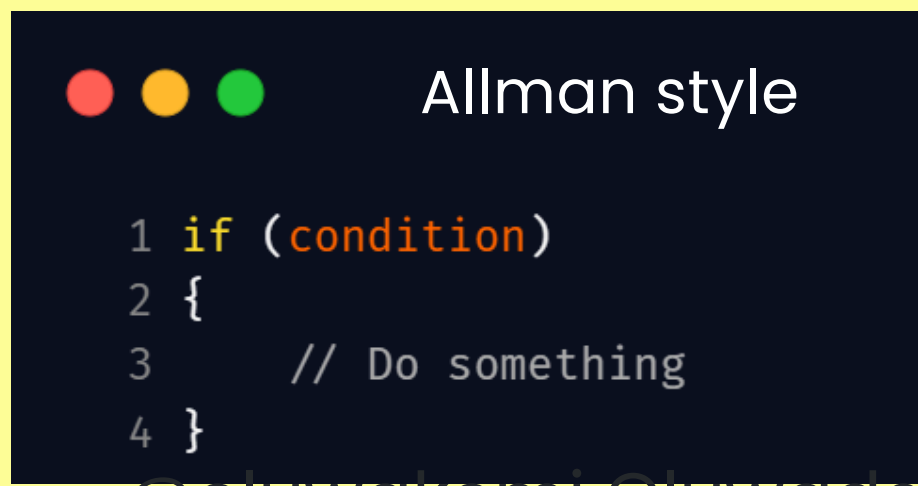
```
1 function greet(name) {  
2     if (name) {  
3         console.log(`Hello, ${name}!`);  
4     }  
5 }
```

- **Brace Style:** Stick to a consistent brace style, such as **K&R** style (opening brace on the same line) or **Allman** style (opening brace on a new line), choose one style and stick to it



K&R style

```
1 if (condition) {  
2     // Do something  
3 }
```



Allman style

```
1 if (condition)  
2 {  
3     // Do something  
4 }
```

- **Quotes:** Stick to either single quotes (') or double quotes (") for strings, and use them consistently throughout your codebase.



```
1 const message = 'Hello, world!';  
2 const userName = "Oluwakemi"
```

- **Consistent Spacing:** Add spaces or empty lines for readability, such as around operators, between functions, and after commas.



low readability

```
1 function greet() {  
2     console.log("Hello");  
3     let name = "John";  
4     return name;  
5 }  
6 let message = "Welcome";  
7 function sayHello() {  
8     console.log(message);  
9 }
```



high readability

```
1 function greet() {  
2     console.log("Hello");  
3     let name = "John";  
4     return name;  
5 }  
6  
7 let message = "Welcome";  
8  
9 function sayHello() {  
10    console.log(message);  
11 }
```

- **Tools:** Use formatting tools like **Prettier** or linting tools like **ESLint** to automate consistent formatting.



### 3. Write Small, Focused Functions

What this means is breaking down your logic into bite-sized, reusable functions, each with a **single responsibility**.

A function should handle one specific task. This keeps it modular and reusable, making it easier to debug and test.


Aim for functions that fit within **20–30 lines of code**, though shorter is often better. If a function grows too large, identify parts that can be extracted into helper functions.

If you have a complex task, break it into smaller steps and write a function for each step. Then, compose those functions to achieve the larger goal.

Let's see an example:



```
1 function processOrder(order) {  
2   if (!order.items.length) {  
3     console.error("No items in the order.");  
4     return;  
5   }  
6   let total = 0;  
7   for (const item of order.items) {  
8     total += item.price;  
9   }  
10  console.log(`Total: ${total}`);  
11  console.log("Order processed.");  
12 }
```

A function   
handling different  
tasks at the same  
time



```
1 function validateOrder(order) {  
2   if (!order.items.length) throw new Error("No items in the order.");  
3 }  
4  
5 function calculateOrderTotal(items) {  
6   return items.reduce((total, item) => total + item.price, 0);  
7 }  
8  
9 function processOrder(order) {  
10  try {  
11    validateOrder(order);  
12    const total = calculateOrderTotal(order.items);  
13    console.log(`Total: ${total}`);  
14    console.log("Order processed.");  
15  } catch (error) {  
16    console.error(error.message);  
17  }  
18 }
```



Each task is being  
handled by  
separate  
functions, making  
it more readable  
and maintainable.

## 4. Use Modern JavaScript (ES6+) Features


Modern JavaScript (ES6 and beyond) introduces features that make your code cleaner, more concise, and easier to understand.

These features not only improve readability but also enhance maintainability and performance.

### Some Modern Javascript (ES6+) Features:

- Simplified Variable Declarations with **let** and **const**


This helps avoid issues with **hoisting** and accidental reassignments caused by **var**.



```
1 // Old way
2 var name = "Oluwakemi";
3
4 // Modern way
5 const age = 30; // Use for values that don't change
6 let city = "Lagos"; // Use for values that can change
```

- **Arrow Functions** for Cleaner Syntax


Arrow functions provide a **shorter**, cleaner syntax for writing functions and lexically bind **this**, making them ideal for callbacks and concise one-liners.



```
1 // Old way
2 function add(a, b) {
3   return a + b;
4 }
5
6 // Modern way
7 const add = (a, b) => a + b; // Shorter and concise
```

- **Template Literals** for Readable Strings

Makes **string concatenation** more intuitive, especially when embedding variables or multi-line strings.



```
1 // Old way
2 const message = "Hello, " + name + "! You are " + age + " years old.";
3
4 // Modern way
5 const message = `Hello, ${name}! You are ${age} years old.`;
```

- **Destructuring for Extracting Values**

Object destructuring allows you to take specific fields from an object and assign them to a variable instantly. It reduces the number of code lines we need to extract the object properties and makes your code easier to understand.

```
1 // Old way
2 const person = { name: "Oluwakemi", age: 30 };
3 const name = person.name;
4 const age = person.age;
5
6 // Modern way
7 const { name, age } = person;
```

- **Spread and Rest Operators** for Flexibility

Handle arrays and objects more elegantly, especially when merging or copying data.

```
1 // Old way
2 const arr1 = [1, 2];
3 const arr2 = [3, 4];
4 const combined = arr1.concat(arr2);
5
6 // Modern way
7 const combined = [...arr1, ...arr2];
```

- **Promises** and **async/await** for Cleaner Asynchronous Code

Handle asynchronous operations without nested callbacks or callback hell, making the code easier to follow.



```
1 // Old way (Callback hell)
2 fetchData((data) => {
3   process(data, (result) => {
4     console.log(result);
5   });
6 });
7
8 // Modern way
9 const fetchData = async () => {
10   const data = await fetch("/api/data");
11   const result = await process(data);
12   console.log(result);
13 };
```

There are other ES6 features, you can read about them for further studies:

- **Modules for** Better Code Organization
- **Optional Chaining** and **Nullish Coalescing** for Safe Access
- **Default Parameters** for Simplifying Function Arguments



## 5. DRY (Don't Repeat Yourself)

"Don't Repeat Yourself" (DRY) is a fundamental principle in software development that emphasizes reducing duplication in code.

The idea is simple: **every piece of knowledge or logic in a program should have a single, unambiguous representation.**

By following the DRY principle, your JavaScript code becomes:

- **Easier to maintain:** Changes need to be made in only one place, reducing errors.
- **More readable:** Clean and concise code is easier to understand.
- **Less error-prone:** Eliminating redundancy reduces the chances of introducing inconsistencies.

## Tips For Avoiding DRY:

- **Use Functions for Reusable Code:** When you notice repeated blocks of logic, encapsulate them in a function. For example:

A code editor window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. The text "Repeated lines of code" is written in white in the top right corner. A large red 'X' is drawn over the top right corner of the editor. The code inside the editor consists of three lines of JavaScript: 

```
1 console.log("Hello, Oluwakemi!");  
2 console.log("Hello, John!");  
3 console.log("Hello, Jane!");
```


Repeated lines of code

A code editor window with a dark blue background and three colored window control buttons (red, yellow, green) in the top left corner. The text "Reusable function is created to achieve the same purpose" is written in white in the top right corner. A large green checkmark is drawn over the top right corner of the editor. The code inside the editor consists of six lines of JavaScript: 

```
1 function greet(name) {  
2     console.log(`Hello, ${name}!`);  
3 }  
4 greet("Oluwakemi");  
5 greet("John");  
6 greet("Jane");
```

Reusable function is created to achieve the same purpose

- **Use Variables for Repeated Values:** Avoid repeating the same values or strings multiple times; instead, store them in variables.



```
1 console.log("Error: Invalid input");
2 console.error("Error: Invalid input");
```




```
1 const errorMessage = "Error: Invalid input";
2 console.log(errorMessage);
3 console.error(errorMessage);
```

- **Leverage Loops:** Replace repetitive tasks with loops to iterate over data structures.




```
1 console.log("Item 1");
2 console.log("Item 2");
3 console.log("Item 3");
```




```
1 const items = ["Item 1", "Item 2", "Item 3"];
2 items.forEach(item => console.log(item));
```

- **Use Objects or Arrays for Related Data:** Combine related data into arrays or objects to simplify management.



```
1 const firstName = "Oluwakemi";  
2 const lastName = "Oluwadahunsi";  
3 console.log(`Full Name: ${firstName} ${lastName}`);
```



```
1 const firstName = "Oluwakemi";  
2 const lastName = "Oluwadahunsi";  
3 console.log(`Full Name: ${firstName} ${lastName}`);
```

I hope you found this material  
useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be  
helpful to someone 🙌

# Hi There!

## Thank you for reading through

Did you enjoy this knowledge?

 Follow my LinkedIn page for more work-life balancing and Coding tips.

 LinkedIn: Oluwakemi Oluwadahunsi

[kodemaven-portfolio.vercel.app](https://kodemaven-portfolio.vercel.app)