In search of Clean Code - its existence and limits

What is Clean Code ?

"Clean Code" is a computer code that is easy to Read, Maintain, and Understand. As human beings, we are acceptable to others when we are simple and expressive to others. Similarly, our code should be simple, concise, and expressive.

However, Clean code is more than just aesthetically pleasing formatting. Clean code reduces the chances of bugs, improves collaboration among developers, and ultimately leads to a more sustainable codebase.

The 4 pillars of Clean Code

1. Readability

Code should be written in a way that is easy for others (or your future self) to read. Meaningful variable names, consistent formatting, and well organised code contribute to readability.

```
// Not-so-clean code
const x = y + z / 3.14;

// Clean code
const radius = diameter / Math.PI;
```

2. Maintainability

A clean codebase is easy to maintain. This involves modularising your code, minimising dependencies, and adhering to design principles like SOLID.

```
// Hard-to-maintain code
function calculateArea(radius) {
    // ...lots of code...
    // ...more code...
    // ...even more code...
    return result;
}

// Maintainable code
function calculateArea(radius) {
    return Math.PI * radius * radius;
}
```

3. Testability

Clean code is inherently testable. It encourages the use of unit tests, making it easier to identify and fix issues.

```
// Untestable code
function getRandomNumber() {
    return Math.random();
}

// Testable code
function getRandomNumber() {
    return 0.5; // Always returns 0.5 for testing consistency
}
```

4. Scalability

As your project grows, clean code scales with it. It accommodates changes and additions without causing a cascade of issues.

```
// Not scalable code
if (condition) {
    // ...do something...
} else {
    // ...do something else...
}
```

```
// Scalable code
function handleCondition(condition) {
    // ...do something...
}
```

Exploring the limits

While clean code is a noble pursuit, are there limits to how clean we can make our JavaScript? Let's explore a few scenarios where maintaining cleanliness might seem challenging.

1. Performance vs. Readability

One common dilemma is the trade-off between performance and readability. For example, inline code might be more performant, but it sacrifices readability.

```
// Less readable but potentially more performant
for (let i = 0; i < array.length; i++) array[i] *= 2;

// More readable but potentially less performant
array = array.map(item => item * 2);
```

The key is to strike a balance based on the context. Always favor readability unless performance is a critical concern.

2. Over-Engineering

Over-engineering occurs when developers add unnecessary complexity to their code. This can lead to bloated codebases that are hard to maintain.

```
// Over-engineered solution

class MathHelper {
    static add(a, b) {
        return a + b;
    }

    static multiply(a, b) {
```

```
        return a * b;
    }
}

const result = MathHelper.multiply(MathHelper.add(3, 5), 2);
```

Simplicity is often the best approach. Only introduce complexity when it's justified.

One more:

```
// Over-engineered code to calculate the sum of an array of numbers

function calculateSum(arr) {
  // Check if arr is an array
  if (Array.isArray(arr)) {
    // Use a for...in loop to iterate through the array
    let sum = 0;
    for (let index in arr) {
      // Check if the current element is a number
      if (!isNaN(arr[index])) {
        // Convert the element to a number (just in case it's a string)
        const num = Number(arr[index]);
        // Add the number to the sum
        sum += num;
      }
    }
    // Return the sum
    return sum;
  } else {
    // Handle the case when the input is not an array
    throw new Error('Input is not an array');
  }
}

// Usage
const numbers = [1, 2, 3, 4, 5];
const sum = calculateSum(numbers);
console.log(sum); // Outputs 15
```

3. Premature Optimization

The famous quote by Donald Knuth rings true: "Premature optimization is the root of all evil." Focusing on optimisation before identifying bottlenecks can lead to code that is hard to understand and maintain.

```
// Prematurely optimised code for summing the numbers in an array

function sumArray(arr) {
```

```
  let sum = 0;
  const length = arr.length; // Store the length of the array
  for (let i = 0; i < length; i++) {
    sum += arr[i];
  }
  return sum;
}

// Usage
const numbers = [1, 2, 3, 4, 5];
const result = sumArray(numbers);
console.log(result); // Outputs 15
```

Optimise when necessary, not just for the sake of optimisation.

Case Studies

Let's examine a couple of case studies to illustrate the principles of clean JavaScript code.

Case Study 1 - Poor Naming

Would we like it if our parents had given us names like a, b123, foo? :)

We must have been furious or looking for a change! Treat the "naming" in a similar way for variables and functions in your code:

```
// Example 1: Poor Naming
function calculate(a, b) {
  const BASE = 10;
  const total = a + (a * (b / 100)) + BASE_TAX;
  console.log(total);
}

calculate(50, 20);
```

Instead

```
// Example 1: Good Naming
function calculateTotalWithTax(basePrice, taxRate) {
  const BASE_TAX = 10;
  const totalWithTax = basePrice + (basePrice * (taxRate / 100)) + BASE_TAX;
  console.log(totalWithTax);
```

```
}

calculateTotalWithTax(50, 20);
```

Case Study 2 - The Callback Pyramid

Consider the infamous "callback pyramid" in asynchronous JavaScript code.

```
getData((data1) => {
    getMoreData(data1, (data2) => {
        getEvenMoreData(data2, (data3) => {
            // ...and so on
        });
    });
});
```

This callback hell is not only hard to read but also prone to errors. A cleaner approach is to use Promises or async/await.

```
async function fetchData() {
    try {
        const data1 = await getData();
        const data2 = await getMoreData(data1);
        const data3 = await getEvenMoreData(data2);
        // ...and so on
    } catch (error) {
        console.error(error);
    }
}
```

Case Study 3 - Magic numbers

Magic numbers, hard-coded constants in your code, can make it difficult to understand the significance of certain values.

```
// Magic number
function calculateArea(radius) {
    return 3.14159 * radius * radius;
}

// Named constant
const PI = Math.PI;

function calculateArea(radius) {
    return PI * radius * radius;
}
```

By using named constants, your code becomes more expressive and self-documenting.

Case Study 4 - Single Source of Truth

Find a single place to store data or configurations of an application. Duplicating them in multiple places will trigger chances of error.

```
// No Single Source of Truth

// File userAPI.js
const API_KEY = '12bbnytg54j';

function async getUserData() {
  // uses the API key to fetch the user data.
}

// File departmentAPI.js
const API_KEY = '12bbnytg54j';

function async getDepartmentData() {
  // uses the API key to fetch the department data.
}
```

Here API key is duplicated in multiple places. You may forget to update them everywhere if they need an update.

They Say Clean Code is SUBJECTIVE

Yes, it could be. something as a clean code to you may not be clear to me as I may not need some of the standards and practices you follow and vice versa. Also, it may be impossible to achieve a 100% clean in a larger project.

Take an example of `Pure Function`

It is a function that produces the same output for the same input.

```
function sayGreeting(name) {
  return `Hello ${name}`;
}
```

A pure function shouldn't have any `side effects` to change the expected output. Is it a pure function?

```
let greeting = "Hello";

function sayGreeting(name) {
  return `${greeting} ${name}`;
}
```

Well, No. The function's output now depends on an outer state called `greeting`. What if someone changes the value of the `greeting` variable to `Hola`? It will change the output of the `sayGreeting()` function.

So, Can I make all functions `Pure Functions`?

Yes, technically, you can. However, the application with only pure functions may not do much.

Your application program will have side effects like `HTTP calls`, `logging to console`, `IO operations`, and many more. Please use pure functions in as many places as you find possible. Isolate impure functions(side effects) as much as possible. It will improve your program's readability, debuggability, and testability a lot.

Conclusion

Clean JavaScript code is not a mythical concept but a practical approach to writing code that is sustainable and efficient. By embracing readability, maintainability, testability, and scalability, developers can navigate the challenges of code cleanliness. While there may be

moments where the pursuit of cleanliness seems to clash with other priorities, understanding the context and finding a balance is key.

In the end, the quest for clean JavaScript code is an ongoing journey—one that requires constant reflection, learning, and adaptation. As we explore its limits and question its existence, we pave the way for a codebase that not only works but also thrives in the ever-evolving landscape of web development.