Programming JS

Margit Tennosaar

JavaScript writing rules

- It is essential to write clean, maintainable, and error-free code

- It helps improve readability, reduce bugs, and ensures better maintainability


Sometimes my code
is like this.....
Don't know, what it does.
But i am scared to delete.

Use "use strict"

- Enables stricter error handling and catches common mistakes early.

- Prevents the use of problematic features.

```
"use strict";
function exampleFunction() {
  // Strict mode applies here
}
```

Consistent formatting

- Indentation: Use 2 or 4 spaces consistently.
- Semicolons: Always use semicolons to avoid unexpected errors.
- Curly Braces: Always use {} even for single-line blocks.

```
if (isActive) {
  console.log("Active");
}
```

Naming conventions

- Variables & Functions: camelCase (e.g., userName)
- Classes: PascalCase (e.g., UserProfile)
- Constants: UPPER_CASE (e.g., MAX_VALUE)

```
const MAX_SPEED = 120;
function calculateDistance() {}
class Vehicle {}
```

Declare variables properly

- Use let for changeable values, const for constants.

- Avoid var to prevent scoping issues.

```
const MAX_USERS = 100;
let currentUser = "Alice";
```

Functions

- Single Responsibility Principle: A function should do one thing (Pure functions)
- Limit parameters: Use objects for multiple parameters.
- Use arrow functions where appropriate.

```
const getUserInfo = (user) => `${user.name} is ${user.age} years old.`;
```

Commenting and documentation

- Write meaningful comments explaining "why," not "what."

- Use JSDoc for better documentation.

```
/**
 * Calculate the area of a rectangle.
 * @param {number} width
 * @param {number} height
 * @returns {number} Area
 */
function calculateArea(width, height) {
  return width * height;
}
```

Avoid magic numbers and strings

```
// Bad
let discount = total * 0.2;

// Good
const DISCOUNT_RATE = 0.2;
let discount = total * DISCOUNT_RATE;
```

Refactor regularly and keep learning

- Improve code structure, readability, and performance.

- Stay updated with JavaScript best practices.

```
// Before refactoring
function add(a, b) { return a + b; }

// After refactoring
const add = (a, b) => a + b;
```

Error handling in JS

Why error handling is important?

- Helps identify and fix issues in your code.
- Prevents programs from crashing unexpectedly.
- Improves the debugging process, making code more reliable and maintainable.

What is wrong?

```
function getOddYears(years) {
  return yeas.filter((year) => year % 2 !== 0);
}
```

Debugging

```
function getOddYears(years) {
  console.log(years); // Check the input
  const oddYears = years.filter((year) => year % 2 !== 0);
  console.log(oddYears); // Check the output
  return oddYears;
}
```

```
function getOddYears(years) {
  debugger; // Execution will pause here
  return years.filter((year) => year % 2 !== 0);
}
```

Try…catch

Handling runtime errors

```
function getOddYears(years) {
  try {
    return years.filter((year) => year % 2 !== 0);
  } catch (error) {
    console.error('An error occurred:', error);
  }
}
```

```
function divideNumbers(a, b) {
  try {
    // Try block: attempt the division
    let result = a / b;  // If b is undefined, this will result in NaN (Not-a-Number)

    if (isNaN(result)) {
      console.log("Invalid operation: Division resulted in NaN");
    } else {
      console.log(`The result is: ${result}`);
    }
  } catch (error) {
    // Catch block: handle any runtime errors (though in this case, no throw is required)
    console.error(`An error occurred: ${error.message}`);
  } finally {
    // Finally block: this always runs, regardless of whether an error occurred or not
    console.log("Execution completed.");
  }
}

// Example calls
divideNumbers(10, 2);  // Valid division
divideNumbers(10);     // Division by undefined (resulting in NaN)
```

Error types in JavaScript

- SyntaxError: Incorrect syntax.
- ReferenceError: Accessing an undefined variable.
- TypeError: Using incorrect data types.
- RangeError: Value outside the expected range.
- URIError: Improper usage of URI functions.

Creating custom errors

- Create custom errors when necessary for better debugging.

```
function checkNumber(num) {
  if (isNaN(num)) {
    throw new Error('Input must be a number');
  }
  console.log('The number is', num);
}

try {
  checkNumber('test'); // Throws an error because 'test' is not a number
} catch (error) {
  console.error(error.message); // Logs "Input must be a number"
}
```