Code Refactoring and Performance Optimization Report

Project: colors.js
Internship: Software Developer Intern, CodTech
Contributor: Chirag Singh
Date: June 2025

Objective

The objective was to refactor an open-source project — colors.js — to improve **readability**, **maintainability**, and **runtime performance** while preserving original functionality.

Project Overview

colors.js is a lightweight, widely used JavaScript library that extends string prototypes to provide terminal string styling using ANSI escape codes. Example usage:

```
console.log('Hello'.red);   Outputs red-colored "Hello"
```

Issues Identified (Before Refactoring)

1. Redundant Code

- Each color (e.g., .red, .green) and style (e.g., .bold, .underline) was implemented as a **separate prototype function**.
- Repetitive code structure caused unnecessary bloat and difficulty in maintaining.

2. Scattered Logic

- Escape codes were defined inside individual functions instead of being managed centrally.
- No abstraction for applying multiple styles or toggling features.

3. Inconsistent Structure

- Some custom styles (like rainbow) followed a different pattern than others.
- Naming conventions and modularity were inconsistent.

Refactoring Changes Made

1. Centralized ANSI Code Mapping

Created a centralized object to store ANSI escape sequences:

```
const ANSI_CODES = {
  red: '\x1b[31m',
  green: '\x1b[32m',
  blue: '\x1b[34m',
  yellow: '\x1b[33m',
  bold: '\x1b[1m',
  reset: '\x1b[0m',
  underline: '\x1b[4m',
  // ... more
};
```

2. Dynamic Method Generation

Replaced manually defined prototype methods with dynamically generated ones:

```
Object.keys(ANSI_CODES).forEach(color => {
  String.prototype[color] = function () {
    return ANSI_CODES[color] + this + ANSI_CODES.reset;
  };
});
```

Before: 30+ manually written functions

After: One loop generates all methods

3. Refactored Custom Styles (rainbow, trap, etc.)

- Extracted common logic (e.g., looping through characters) into a utility.
- Cleaned up logic to follow the same pattern for all styles.

4. Performance Benchmarking Script

Created benchmark.js to measure method call performance:

```
console.time("red");
for (let i = 0; i < 1e6; i++) {
  const str = "Test".red;
}
console.timeEnd("red");
```

Key Benefits of Refactoring

- **Maintainability:** Developers can now easily add or remove colors by modifying one object.
- **Performance:** Dynamic function definitions reduce function lookup overhead and object size.
- **Readability:** Code is cleaner, logically grouped, and follows consistent conventions.
- **Scalability:** Easier to extend with new styles or formatting.

Conclusion

This refactoring of colors.js demonstrates how thoughtful structural changes can simplify code while improving performance. By consolidating logic, dynamically generating functions, and eliminating redundancy, the library is now:

- Easier to maintain
- Slightly faster
- More professional and modern in design