

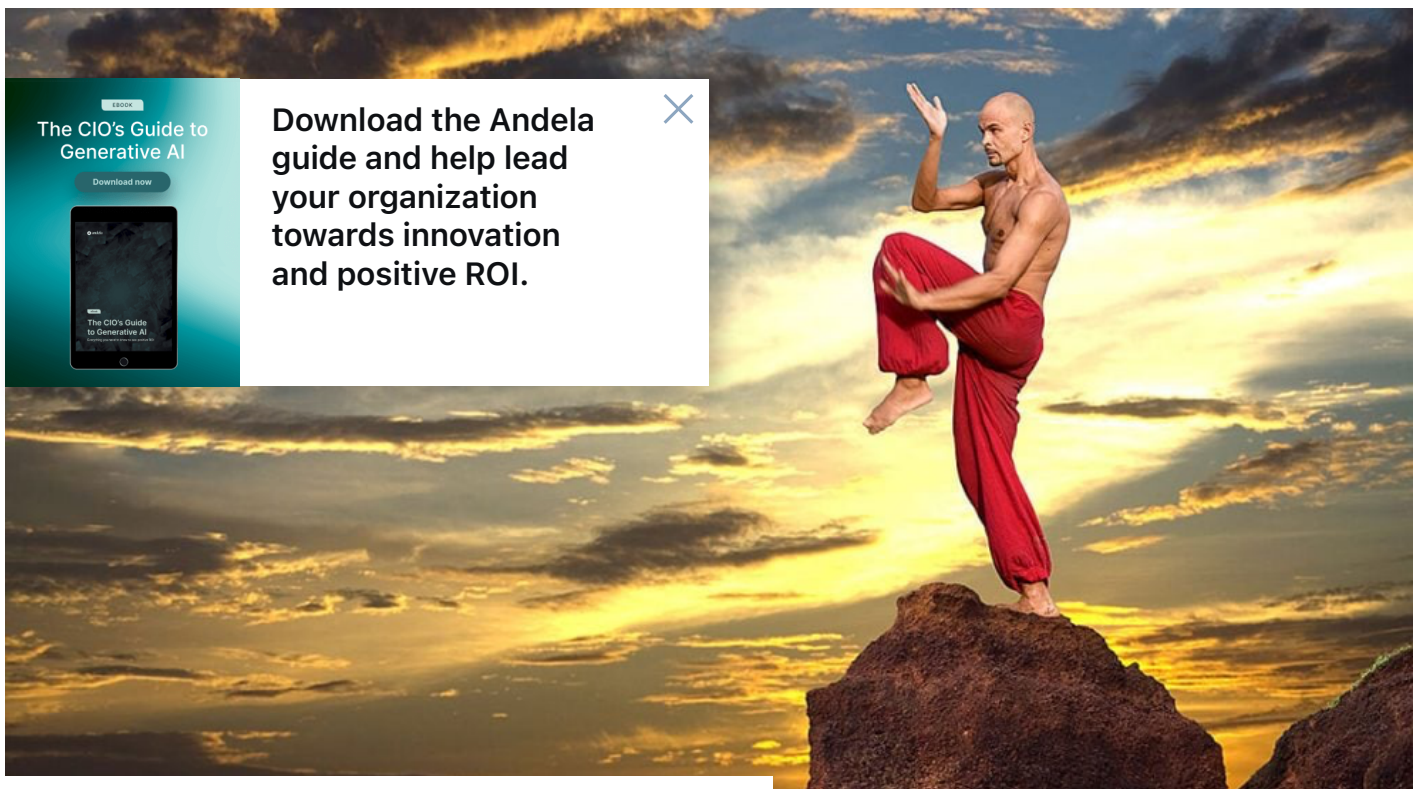


CI/CD / JAVASCRIPT

# JavaScript Kung Fu: Elegant Techniques To Master the Language

Whether you're building enterprise software or just honing your skills, these patterns will help you write cleaner, faster and more maintainable code.


Jun 12th, 2025 7:00am by [Zziwa Raymond Ian](#)



BOOK

The CIO's Guide to Generative AI

Download now



Download the Andela guide and help lead your organization towards innovation and positive ROI.

X

Andela sponsored this post.

JavaScript isn't just a **language** — it's a specialized craft. And like any

skilled martial art, the difference between being a novice and a master lies in the techniques you perfect. Anyone can write code that works, but writing code that's clean, powerful and elegantly expressive is where **true JavaScript Kung Fu** begins.

Let's deep dive into **elevated JavaScript techniques** that will make your code stand out from the crowd. It doesn't matter if you're building enterprise software or just sharpening your skills, these patterns will help you write cleaner, faster and more maintainable code — the kind that will make your future self (and teammates) nod with respect.

We'll cover everything from:

- Smart `console.log` practices that go beyond the basics
- Using **ES6 features** like template literals, destructuring and spread operators for clearer, more intuitive logic
- Writing more expressive conditionals with **short-circuit** evaluation
- Using tools like `console.table()` and `console.time()` for debugging and performance profiling
- And combining arrays like a boss, even with duplicates

Think of this as your dojo for **modern JavaScript mastery**. You've got the basics, but you're ready to take it up a level.

Let's code like a sensei.



Andela provides the world's largest private marketplace for global remote tech talent driven by an AI-powered platform to manage the complete contract hiring lifecycle. Andela helps companies scale teams & deliver projects faster via specialized areas: App Engineering, AI, Cloud, Data & Analytics.

[Learn More →](#)

#### THE LATEST FROM ANDELA

### [Andela | The Rise of the AI Engineer: Why Coding Isn't Dead, It's Evolving](#)

11 June 2025

### [Andela | Andela at Safaricom Decode 2025: An Inspiring Journey into Africa's Tech Revolution](#)

4 June 2025

### [Andela | Critical Programming: Your Next Teammate Is AI and You're the Conductor](#)

2 June 2025

## 1. `console.log` Techniques: Logging Like a Pro

In the early stages of JavaScript development, many developers rely heavily on basic `console.log()` statements. While it gets the job done, advanced developers know there are far more powerful and cleaner ways to log data, especially when working with structured objects, debugging or profiling performance. Consider this simple example:

```
1 const student = { name: 'Ray', age: 20 };  
2 const tutor = { name: 'Peter', age: 40 };
```

#### TRENDING STORIES

1. [Convert a Google Spreadsheet to JSON-Formatted Text](#)
2. [5 Technical JavaScript Trends You Need To Know About in](#)

2025

3. [Recreating Shopify's BFCM Globe Using react-globe.gl](#)
4. [JavaScript Framework Reality Check: What's Actually Working](#)
5. [5 Ways JavaScript Is Improving Modules for Developers](#)

## Basic Logging (Harder to Read)

```
1 console.log(student);  
2 console.log(tutor);
```

While this works, it's harder to distinguish between the logs, especially when dealing with multiple variables. Now, let's explore cleaner alternatives.

## 2. Computed Property Logging

Instead of logging variables one by one, you can use shorthand object property names to group your logs into a single object, making them more readable:

```
1 console.log({ student, tutor });  
2  
3 //This prints  
4 { student: { name: 'Ray', age: 20 }, tutor: { name: 'Peter', age: 40 } }
```

Now the log is self-explanatory, clear and easier to inspect — perfect for debugging or reviewing nested data structures.

## 3. Custom Styling With %c

For UI-focused logs or when you want to highlight something specific

in the console, you can use the `%c` directive along with CSS styles:

```
1 console.log('%c student', 'color: orange; font-weight: bold');
```

This is especially useful in large applications where you need to visually separate logs from one another, such as highlighting warnings, steps or critical state updates.

## 4. Displaying Objects in a Table With `console.table()`

When dealing with arrays of objects (or even a single object with multiple similar keys), `console.table()` is a visually superior alternative:

```
1 console.table([student, tutor]);
```

This logs a structured table with rows and columns, making it easy to scan and compare values. It's ideal when you're working with datasets like API responses, lists of users or status objects.

## 5. Benchmarking With `console.time()` and

`console.timeEnd()`

Performance profiling is critical in JavaScript, especially when loops or operations may impact responsiveness. Use `console.time()` and `console.timeEnd()` to measure execution time:

```
1 console.time('loop');  
2  
3 let i = 0;  
4 while (i < 1000000) {  
5   i++;  
6 }
```

```
7
8 console.timeEnd('loop');
9
10 //This will print something like
11 loop: 5.384ms
```

This gives you insight into how long your logic takes to execute, helping with optimization decisions or detecting bottlenecks in your code.

By mastering these enhanced logging techniques, you're not just printing data — you're communicating intent, improving readability and debugging smarter. These tools are part of your JavaScript utility belt, helping you become more efficient, organized and effective in your development workflow.

## Template Literals: Writing Strings With Superpowers

Gone are the days of messy string concatenation using the + operator. With template literals, introduced in ES6, JavaScript gives you a cleaner, more expressive way to build strings, especially when variables and expressions are involved. Template literals are enclosed in backticks instead of single or double quotes, and they support interpolation, multiline strings and even embedded expressions.

Let's break it down:

```
1 //Basic String Interpolation
2 const name = 'Ray';
3 const age = 20;
4
5 console.log(`My name is ${name} and I am ${age} years old.`);
```

Instead of writing:

```
'My name is ' + name + ' and I am ' + age + ' years old.'
```

Template literals let you inject variables directly into your string using `${}` . This not only reduces clutter but also boosts readability and maintainability.

```
1 // Multiline Strings (No \n Needed!)
2 const message = `Hello,
3 This is a multiline message,
4 Neatly written with template literals.`;
5
6 console.log(message);
```

Previously, you'd have to use `\n` or concatenate lines awkwardly. With backticks, you just write across lines naturally.

```
1 //Expressions and Function Calls Inside Template Literals
2 const price = 100;
3 const discount = 0.2;
4
5 console.log(`Final price after discount: ${price - (price * discount)} UGX`);
```

You can even call functions inside:

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

Template literals make your code more expressive and cleaner,

especially in scenarios involving dynamic data, API responses or logging. Whether you're crafting UI messages, generating reports or debugging output, this feature adds elegance and power to your strings.

## Using `reduce()` To Get Totals: From Clunky Loops To Clean Logic

When working with arrays of numbers or objects, it's common to calculate a total, be it prices, scores or item quantities. Many developers fall into the trap of using verbose `for` loops or `forEach()` to accomplish this. While those methods work, they're often less expressive and harder to maintain. The `reduce()` method offers a powerful and elegant alternative that not only condenses your code but also enhances its readability and intent.

```
1 // Bad Code - Using forEach() for Totals
2 const cart = [
3   { item: 'Book', price: 15 },
4   { item: 'Pen', price: 5 },
5   { item: 'Notebook', price: 10 },
6 ];
7
8 let total = 0;
9
10 cart.forEach(product => {
11   total += product.price;
12 });
13
14 console.log(`Total price: ${total} UGX`);
```

This approach works, but it's verbose, especially in large codebases. You're manually managing the total, which can lead to errors and side effects.



```
1 const cart = [
2   { item: 'Book', price: 15 },
3   { item: 'Pen', price: 5 },
4   { item: 'Notebook', price: 10 },
5 ];
6
7 const total = cart.reduce((acc, curr) => acc + curr.price, 0);
8
9 console.log(`Total price: ${total} UGX`);
```

## Here's why `reduce()` is superior:

- `acc` (accumulator) starts at `0` (the second argument of `reduce()` ).
- On each iteration, it adds `curr.price` to `acc` .
- After the loop finishes, you get the final total — clean, functional and declarative.

You can even make it more expressive with arrow functions and default values:

```
1 const total = cart.reduce((sum, { price }) => sum + price, 0);
```

This version uses destructuring to directly access the price of each item, making the code more readable.

So next time you're summing values in an array, ditch the loop and reach for `reduce()` . It's the functional way to write logic that's not just correct, but clean and powerful.

## Understanding the Spread Operator and Array Merging

The spread operator (...) is one of the most elegant and versatile

features introduced in ES6. It allows you to “spread” the elements of an iterable (like arrays or objects) into individual elements. This comes in handy especially when merging arrays, cloning or passing multiple values in a function call.

## Basic Array Merging (Clean and Modern Way)

Let’s say you have two arrays:

```
1 const arr1 = [1, 2, 3];  
2 const arr2 = [4, 5, 6];
```

Here’s the old-school approach using `concat()` :

```
1 const merged = arr1.concat(arr2);  
2 console.log(merged); // [1, 2, 3, 4, 5, 6]
```

This works, but it’s a bit verbose and less readable compared to modern syntax.

And here’s the modern approach using the spread operator:

```
1 const merged = [...arr1, ...arr2];  
2 console.log(merged); // [1, 2, 3, 4, 5, 6]
```

The new method is much cleaner, more expressive and easier to understand. You’re literally just saying: “Spread the values of `arr1` and `arr2` into one new array.”

In case of duplicates, merge with `set` and the spread operator.

Let's assume the arrays have duplicate values:

```
1 const arr1 = [1, 2, 3, 3];
2 const arr2 = [3, 4, 5, 6, 6];
```

If we merge them directly:

```
1 const merged = [...arr1, ...arr2];
2 console.log(merged); // [1, 2, 3, 3, 3, 4, 5, 6, 6]
```

We now have duplicate values. In many cases (like IDs, tags, etc.), we want unique values.

So to remove the duplicate values, we can use `Set` and the spread operator.


```
1 const uniqueMerge = [...new Set([...arr1, ...arr2])];
2 console.log(uniqueMerge); // [1, 2, 3, 4, 5, 6]
```

To give you context, `Set` is a built-in object that only stores unique values. We first use `[...]` to merge the arrays. We then wrap it with `new Set(...)` to filter out duplicates, and finally, we spread that back into a new array.

## Cloning an Array With Spread

If you ever want to copy an array without modifying the original, use the spread operator:

```
1 const original = [10, 20, 30];
2 const copy = [...original];
3
4 copy.push(40);
```

```
5 console.log(original); // [10, 20, 30]  Unchanged
6 console.log(copy); // [10, 20, 30, 40]
```

## Understanding the Relevance of the Rest Operator

The rest operator (...) looks exactly like the spread operator, but it works in the opposite way. While the spread operator expands items, the rest operator gathers items into a single array or object.

It's used mainly in:

1. Function arguments
2. Array destructuring
3. Object destructuring

### 1. Rest in Function Parameters

Sometimes, you don't know how many arguments a function will receive. Instead of listing them manually, use the rest operator to bundle them up.

```
1 function sum(...numbers) {
2   return numbers.reduce((total, num) => total + num, 0);
3 }
4
5 console.log(sum(1, 2, 3)); // 6
6 console.log(sum(5, 10, 15, 20)); // 50
```

### Explanation:

- `...numbers` collects all the passed arguments into an array.
- You can then manipulate that array like normal (such as using `.reduce()` to add them up).

## 2. Rest in Array Destructuring

The rest operator is handy when you want to get some values individually and gather the rest.

```
1 const colors = ['red', 'green', 'blue', 'yellow'];
2
3 const [primary, secondary, ...others] = colors;
4
5 console.log(primary); // red
6 console.log(secondary); // green
7 console.log(others); // ['blue', 'yellow']
```

### Explanation:

- The first two values are extracted.
- The rest operator gathers everything else into a new array, `others`.

## 3. Rest in Object Destructuring

You can also extract specific properties from an object and collect the remaining ones.

```
1 const user = {
2   id: 1,
3   name: 'Ray',
4   age: 25,
5   role: 'developer'
6 };
7
8 const { name, ...rest } = user;
9 console.log(name); // 'Ray'
10 console.log(rest); // { id: 1, age: 25, role: 'developer' }
```

### Explanation:

We pull out the name property.

The rest operator collects the remaining key-value pairs into `rest`.

While the rest (...) and spread (...) operators look identical in syntax, their behavior and purpose are fundamentally different. The spread operator is used to expand an array or object into individual elements or properties. You'll typically see it used when passing arguments into a function or when combining arrays and objects. In contrast, the rest operator does the opposite. It's used to gather multiple values into a single array or object. It's commonly applied in function parameters to accept an indefinite number of arguments or in destructuring to collect remaining values. Think of it this way: Spread expands outward, breaking structures apart, while rest gathers inward, bundling them together. This subtle yet powerful distinction is key to writing flexible and clean JavaScript code.

## Understanding Object Data Extraction in JavaScript

In JavaScript, object destructuring is a shorthand syntax that allows you to pull out properties from objects (or elements from arrays) into standalone variables. This makes your code cleaner and reduces redundancy.

Let's take a basic object:

```
1  const me = {  
2    name: 'Ray',  
3    age: 20,  
4    likes: ['football', 'racing']  
5  };
```

## Traditional Way (Less Elegant)

If you want to access values, you'd typically do:

```
1 console.log(me.age); // 20
2 console.log(me.likes[0]); // 'football'
```

While this works just fine, it can become verbose or repetitive, especially when you need to reference the same property multiple times.

## Now Let's Use Destructuring for Cleaner Code

Instead of repeatedly writing `me.age`, you can destructure like this:

```
1 const { age } = me;
2 console.log(age); // 20
```

You can also extract multiple properties at once:

```
1 const { name, likes } = me;
2 console.log(name); // 'Ray'
3 console.log(likes); // ['football', 'racing']
```

## Destructuring With Arrays Inside Objects

Since `likes` is an array, we can destructure it, too:

```
1 const { likes: [first, second] } = me;
2
3 console.log(first); // 'football'
4 console.log(second); // 'racing'
```

Notice how we used an alias. We didn't extract the `likes` array itself, but instead unpacked its elements directly into `first` and `second`.

## Nullish Coalescing (??)

The nullish coalescing operator (`??`) is used to provide a default value when a variable is `null` or `undefined`. It's especially useful when you want to fall back only for truly "empty" values and not for falsy values like `0`, `''` or `false`.

Without nullish coalescing:

```
1 const age = 0;
2 const result = age || 18;
3 console.log(result); // 18 ❌ (0 is falsy, so it falls back)
```

With nullish coalescing:

```
1 const age = 0;
2 const result = age ?? 18;
3 console.log(result); // 0 ✅
```

## Explanation:

- `||` falls back for any falsy value (`0`, `false`, `''`, `null`, `undefined`).
- `??` falls back only for `null` or `undefined`.

## Converting Strings to Integers Using the + Sign

In JavaScript, the unary plus (+) operator can be used to quickly convert strings into numbers.



## Traditional method:

```
1 const numStr = '42';
2 const num = parseInt(numStr);
3 console.log(num); // 42
```

## Shorthand using +:

```
1 const numStr = '42';
2 const num = +numStr;
3 console.log(num); // 42
```

This is a quick and readable trick when you're sure the string contains a valid number. If it doesn't, the result will be `NaN`.

See the code below:

```
1 console.log(+'hello'); // NaN
```

## Optional Chaining (?.)

Optional chaining lets you safely access deeply nested object properties without having to check if each level exists. If the reference is `null` or `undefined`, it returns `undefined` instead of throwing an error.

Without optional chaining:

```
1 const user = {};
2 console.log(user.profile.name); // ❌ Error: Cannot read property 'name' of undefined
```

With optional chaining:

```
1 const user = {};  
2 console.log(user.profile?.name); //  undefined
```

You can also use it with functions:

```
1 user.sayHi?.(); // Only calls sayHi if it exists
```

This is great for handling data from APIs\*\* where some fields may be missing.

## The Ternary Operator

The ternary operator is a shorthand for `if/else`. It follows this syntax:

```
1 condition ? valueIfTrue : valueIfFalse;
```

To give you context, see the code below:

```
1 const age = 20;  
2 const canVote = age >= 18 ? 'Yes' : 'No';  
3 console.log(canVote); // "Yes"
```

This is equivalent to:

```
1 let canVote;  
2 if (age >= 18) {  
3   canVote = 'Yes';  
4 } else {  
5   canVote = 'No';  
6 }  
7 }
```

## Using `||` for Defaults

The logical OR (||) operator can be used to assign default values if the left-hand side is falsy:

```
1 const name = userInput || 'Guest';
2
3 If `userInput` is falsy (like `null`, `undefined`, or ``), `name` will be `Guest`
```

## Logical Short-Circuiting

JavaScript also allows you to write short if statements without `else` using the `&&` operator.

```
condition && expression;
```

For example:

```
1 const isLoggedIn = true;
2 isLoggedIn &&& console.log('User is logged in');
```

It only prints if `isLoggedIn is true`. Equivalent to:

```
1 if (isLoggedIn) {
2   console.log('User is logged in');
3 }
```

## Conclusion

Mastering JavaScript is about writing code that's elegant, efficient and expressive. These techniques, from smarter `console.log()` practices and template literals to the subtle power of destructuring, spread/rest operators and short conditional logic will help you elevate your code from functional to formidable.

Think of these patterns as tools in your martial arts belt, with each one sharpening your ability to write cleaner, faster and more maintainable code. Whether you're debugging like a pro with `console.table()` and `console.time()`, or merging arrays with surgical precision, every skill you build makes your JavaScript Kung Fu stronger.

Now go forth and code like a sensei. Your future self and your teammates will thank you for it.

Want to dive deeper into the possibilities of working with JavaScript? [Read Andela's full guide "Mastering JavaScript Proxies and Reflect for Real-World Use."](#)

TNS

---



Zziwa Raymond Ian is a full-stack engineer and a member of the Andela Talent Network, a private global marketplace for digital talent. Specializing in Next.js, React, JavaScript, TypeScript, NestJs and others, he has developed a deep holistic understanding of both...

[Read more from Zziwa Raymond Ian →](#)

TNS owner Insight Partners is an investor in: [Real.](#)

---

## SHARE THIS STORY



---

## TRENDING STORIES

1. [Convert a Google Spreadsheet to JSON-Formatted Text](#)

2. **5 Technical JavaScript Trends You Need To Know About in 2025**
3. **Recreating Shopify's BFCM Globe Using react-globe.gl**
4. **JavaScript Framework Reality Check: What's Actually Working**
5. **5 Ways JavaScript Is Improving Modules for Developers**

INSIGHTS FROM OUR SPONSOR



Andela provides the world's largest private marketplace for global remote tech talent driven by an AI-powered platform to manage the complete contract hiring lifecycle. Andela helps companies scale teams & deliver projects faster via specialized areas: App Engineering, AI, Cloud, Data & Analytics.

[Learn More →](#)

## **Andela | The Rise of the AI Engineer: Why Coding Isn't Dead, It's Evolving**

11 June 2025

## **Andela | Andela at Safaricom Decode 2025: An Inspiring Journey into Africa's Tech Revolution**

4 June 2025

## **Andela | Critical Programming: Your Next Teammate Is AI and You're the Conductor**

2 June 2025

## **Andela | Vector Databases: Transforming Raw Data into AI Magic**

30 May 2025

## **Andela | From Code to Copilot: How Andela's AI Academy is Reshaping Software Development**

29 May 2025

## **Andela | When Microservices Go Rogue: The Saga Pattern That Keeps Distributed Systems in Line**

27 May 2025

---

**TNS DAILY NEWSLETTER**

**Receive a free roundup of the  
most recent TNS articles in  
your inbox each day.**

EMAIL ADDRESS

SUBSCRIBE

The New Stack does not sell your information or share it with unaffiliated third parties. By continuing, you agree to our [Terms of Use](#) and [Privacy Policy](#).

---

## ARCHITECTURE

Cloud Native Ecosystem  
Containers  
Databases  
Edge Computing  
Infrastructure as Code  
Linux  
Microservices  
Open Source  
Networking  
Storage

## ENGINEERING

AI

AI Engineering  
API Management  
Backend development  
Data  
Frontend Development  
Large Language Models  
Security  
Software Development  
WebAssembly

## **OPERATIONS**

AI Operations  
CI/CD  
Cloud Services  
DevOps  
Kubernetes  
Observability  
Operations  
Platform Engineering

## **CHANNELS**

Podcasts  
Ebooks  
Events  
Newsletter  
TNS RSS Feeds

## **THE NEW STACK**

About / Contact  
Sponsors



## Advertise With Us

### Contributions



Community created roadmaps, articles, resources and journeys for developers to help you choose your path and grow in your career.

Frontend Developer Roadmap

Backend Developer Roadmap

Devops Roadmap

#### FOLLOW TNS



© The New Stack 2025

[Disclosures](#)

[Terms of Use](#)

[Advertising Terms & Conditions](#)

[Privacy Policy](#)

[Cookie Policy](#)