CODING ARTIST

YOUR QUICK REFERENCE GUIDE

JAVASCRIPT CHEAT SHEET

Everything you need to write better JS code, faster!

01 **Variables**
var, let, const and when to use each.

02 **Functions**
Declaration, expression and arrow functions.

03 **Array Methods**
map(), filter(), find(), reduce() and more.

04 **Loops**
for, while, for...of and forEach.

05 **DOM Manipulation**
Select, modify, add, remove elements.

06 **Event Listeners**
click, input, submit and keyboard events.

07 **Async JS**
Promises, fetch API and async/await.

08 **ES6+ Features**
Destructuring, spread, template literals & more.

09 **Local Storage**
setItem, getItem, removeItem and clear.

10 **One-Liners**
Handy code snippets for daily use.

11 **Type Coercion**
== vs === and common conversions.

12 **Best Practices**
Write clean, readable and maintainable code.

Save this post and level up your JavaScript skills!

01 VARIABLES

Variables store data that can be used and updated in your code.

```
let message = "Hello JS!";
const count = 10;
var oldWay = true;
```

TYPES OF VARIABLES

var

The old way of declaring variables (function scoped).

- Function scoped (not block scoped)
- Hoisted and initialized with undefined
- Can be re-declared and updated
- Avoid using in modern JavaScript

```
var name = "John";
var name = "Doe";   // Allowed
name = "Coder";     // Update

// Function scoped
function test() {
  var x = 10;
}
console.log(x); // Works (not ideal)
```

let

Modern way for variables that can change.

- Block scoped
- Hoisted but not initialized (Temporal Dead Zone)
- Can be updated but not re-declared
- Use for values that change

```
let age = 25;
age = 26;       // Update allowed
let age = 30;   // Error

if (true) {
  let score = 100;
}
console.log(score); // Error
// Block scoped
```

const

For values that should NOT change.

- Block scoped
- Hoisted but not initialized (Temporal Dead Zone)
- Cannot be re-declared or re-assigned
- Use for constants and immutable values

```
const PI = 3.14;
const user = { name: "John" };
PI = 3.141;       // Error
user = {};        // Error

// But object/array content
// can be changed
user.name = "Doe"; // Allowed
```

BEST PRACTICES

- Use const by default.
- Use let when the value can change.
- Avoid var – use it only when you must support very old browsers.
- Choose meaningful and descriptive variable names.
- Keep your variables scoped to the smallest possible block.

02 FUNCTIONS

Functions are reusable blocks of code that perform a specific task.

```js
function add(a, b) {
  return a + b;
}

console.log(add(3, 5)); // 8
```

TYPES OF FUNCTIONS

1. FUNCTION DECLARATION

Traditional way of defining a function.

```js
function greet(name) {
  return `Hello, ${name}!`;
}
```

EXAMPLE

```js
greet("John");  // Hello, John!
```

Hoisted: You can call it before it's defined.

2. FUNCTION EXPRESSION

Store a function in a variable.

```js
const greet = function(name) {
  return `Hello, ${name}!`;
};
```

EXAMPLE

```js
greet("John");  // Hello, John!
```

Not hoisted: You can't call it before it's defined.

3. ARROW FUNCTION

Shorter syntax with a more modern look.

```js
const greet = (name) => {
  return `Hello, ${name}!`;
};
```

EXAMPLE

```js
greet("John");  // Hello, John!
```

Does NOT have its own `this`.

ARROW FUNCTION SHORTCUTS

Arrow functions have shorter syntax for simple cases.

SYNTAX	EXAMPLE	SAME AS
`(param) => expression`	`const square = x => x * x;`	`const square = function(x) { return x * x; };`
`() => expression`	`const getPI = () => 3.14159;`	`const getPI = function() { return 3.14159; };`
`(param1, param2) => { }`	`const add = (a, b) => { return a + b; };`	`const add = function(a, b) { return a + b; };`

TIPS

- Use functions to avoid repeating code.
- Use meaningful names.
- Keep functions small and focused.
- Use arrow functions for short callbacks.
- Use declaration for general use.

Functions make your code DRY, readable and easy to maintain.
Write once. Use anywhere.

03 ARRAY METHODS

Powerful built-in methods to work with arrays like a pro! ⚡

```js
const numbers = [1, 2, 3, 4, 5];
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 }
];
```

.map()

Creates a new array with the results of calling a function for each element.

```
numbers.map(n => n * 2);
```

Returns: `[2, 4, 6, 8, 10]`

.filter()

Creates a new array with all elements that pass the test.

```
numbers.filter(n => n % 2 === 0);
```

Returns: `[2, 4]`

.find()

Returns the first element that satisfies the condition.

```
users.find(u => u.age > 25);
```

Returns: `{ name: 'Bob', age: 30 }`

.findIndex()

Returns the index of the first element that satisfies the condition.

```
numbers.findIndex(n => n > 3);
```

Returns: `3`

.reduce()

Reduces the array to a single value.

```
numbers.reduce((acc, n) => acc + n, 0);
```

Returns: `15`

.some()

Checks if at least one element passes the test.

```
numbers.some(n => n > 4);
```

Returns: `true`

.every()

Checks if all elements pass the test.

```
numbers.every(n => n > 0);
```

Returns: `true`

.slice(start, end)

Returns a shallow copy of a portion of an array.

```
numbers.slice(1, 4);
```

Returns: `[2, 3, 4]`

.push()

Adds one or more elements to the end of an array.

```
numbers.push(6);
numbers;
```

Returns new length: `6`

Array: `[1, 2, 3, 4, 5, 6]`

.pop()

Removes the last element from an array.

```
numbers.pop();
numbers;
```

Returns removed item: `6`

Array: `[1, 2, 3, 4, 5]`

QUICK TIP

Array methods don't change the original array (except push(), pop(), shift(), unshift(), splice()).

```
const arr = [1, 2, 3];
const newArr = arr.map(n => n * 2);
console.log(arr);    // [1, 2, 3]
console.log(newArr); // [2, 4, 6]
```

Most of them return new arrays!

Save this cheat sheet for quick reference!

Share with your dev friends!

04 LOOPS

Loops help you repeat code efficiently.

```
const fruits = ['🍎', '🍌', '🍇'];
let text = '';

// Loop through the array
for (let fruit of fruits) {
  text += fruit + ' ';
}
console.log(text); // 🍎 🍌 🍇
```

1. for LOOP

for

Use when you know how many times to loop.

```
for (let i = 0;
    i < 5;;
    i++) {
   console.log(i);
}
```

Output:
0 1 2 3 4

2. while LOOP

while

Use when you don't know how many times to loop.

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

Output:
0 1 2 3 4

3. for...of LOOP

for...of

Use to loop through the values of an array, string, etc.

```
for (let fruit of
  fruits) {
  console.log(fruit);
}
```

Output:
🍎
🍌
🍇
∞

4. forEach()

forEach

Use to loop through each element of an array.

```
fruits.forEach((fruit,
  index) => {
   console.log(index,
   fruit);
});
```

Output:
0 🍎
1 🍌
2 🍇

QUICK TIPS

- Use for loop when you know the count.
- Use while loop when the condition is based on a logic.
- Use for....of for arrays, strings, maps, sets.
- Use forEach() for clean and readable array iteration.
- Avoid infinite loops – always update your condition!

EXAMPLE

```
let sum = 0;
for (let i = 1; i <= 5; i++) {
  sum += i;
}
console.log(sum); // 15
```

Loops make your code SHORTER, SMARTER and POWERFUL!
Write less. Do more.

05 DOM MANIPULATION

Interact with HTML elements and change your web page dynamically.

```js
const title = document.querySelector("h1");
title.textContent = "Hello, DOM!";
title.style.color = "cyan";
title.classList.add("highlight");

// Change content, styles, classes
// Create, add or remove elements
```

SELECTING ELEMENTS

document.getElementById()

Select by ID

```
getElementById("id")
Ex: getElementById("title")
```

document.querySelector()

Select first match

```
querySelector("selector")
Ex: querySelector(".btn")
```

document.querySelectorAll()

Select all matches

```
querySelectorAll("selector")
Ex: querySelectorAll("p")
```

document.getElementsByClassName()

Select by class name

```
getElementsByClassName("class")
Ex: getElementsByClassName("item")
```

MANIPULATING CONTENT

.textContent

Sets or returns plain text.

```
el.textContent = = "Hello";
```

No HTML tags.

.innerHTML

Sets or returns HTML content.

```
el.innerHTML = "<b>Hello</b>";
```

Parses HTML tags.

.value

Sets or returns the value of form elements.

```
input.value = "John";
```

For inputs, selects, etc.

MANIPULATING CLASSES & STYLES

.classList

Add, remove or toggle CSS classes.

```
el.classList.add("active");
el.classList.remove("hide");
el.classList.toggle("show");
```

.style

Change inline styles.

```
el.style.color = "red";
el.style.fontSize = "20px";
el.style.display = "none";
```

CREATING & MODIFYING ELEMENTS

1. createElement()

Creates a new HTML element.

```
const div = document.createElement("div");
```

2. setAttribute()

Sets an attribute on the element.

```
div.setAttribute("id", "box");
div.setAttribute("class", "card");
```

3. appendChild()

Adds the element as a child.

```
document.body.appendChild(div);
parent.appendChild(child);
```

4. remove()

Removes the element.

```
div.remove();
el.parentNode.removeChild(el);
```

QUICK TIPS

- Use querySelector() for modern and flexible selection.
- Modify text with textContent when you don't need HTML.
- Use classList to manage classes efficiently.
- Always keep your DOM clean and simple!

Master the DOM, build interactive and amazing web experiences!

Save this post for quick reference!

Share with your dev friends!

06 EVENT LISTENERS

Listen for events and run code when they happen.

```js
// Add an event listener
element.addEventListener("event", callback);
// Example
button.addEventListener("click", () => {
  console.log("Button clicked!");
});
```

COMMON EVENTS

click
Fires when an element is clicked.

```
button.addEventListener
("cliceb", () => {
  console.log("Clicked");
});
```

dblclick
Fires when an element is double-clicked.

```
element.addEventListener
("dblclick", () => {
  console.log("Double
  Clicked");
});
```

keydown
Fires when a key is pressed down.

```
document.addEventListener
("keydowm", (e) => {
  console.log(e.key);
});
```

keyup
Fires when a key is released.

```
document.addEventListener
("keyup", (e) => {
  console.log(e.key);
});
```

input
Fires when the value of an input changes.

```
input.addEventListener
("input", (e) => {
  console.log(e.target.value);
});
```

submit
Fires when a form is submitted.

```
form.addEventListener
("submit", (e) => {
  e.preventDefault();
  console.log("Form Submitted");
});
```

mouseover
Fires when mouse enters an element.

```
element.addEventListener
("mouseover", () => {
  console.log("Mouse Over");
});
```

mouseout
Fires when mouse leaves an element.

```
element.addEventListener
("mouseoout", () => {
  console.log("Mouse Out");
});
```

THE EVENT OBJECT (e)

Provides useful information about the event.

`e.target`	Element where the event occurred
`e.type`	Type of the event
`e.key`	Key pressed (for keyboard events)
`e.clientX`	Mouse X position
`e.clientY`	Mouse Y position
`e.preventDefault()`	Prevents the default action
`e.stopPropagation()`	Stops the event from bubbling

EVENT LISTENER OPTIONS

You can pass an options object as the third parameter.

```
element.addEventListener("event", callback, options);
```

`capture:`	true to capture the event in the capturing phase.
`once:`	true to remove the listener after it's triggered once.
`passive:`	true to tell the browser you won't preventDefault().
`signal:`	AbortSignal to remove listener using AbortController.

QUICK TIPS

- Use meaningful function names for better readability.
- Remove event listeners when they are no longer needed.
- Use event delegation for better performance on dynamic elements.
- Use `e.preventDefault()` carefully and only when needed.

REMOVE LISTENER

```
function handleClick() {
  console.log("Clicked!");
}
button.addEventListener("click", handleClick);
// Later...
button.removeEventListener("click", handleClick);
```

Save this post for quick reference!

Share with your dev friends!

Follow for more web dev content!

07 ASYNC JAVASCRIPT

Handle async operations like a pro and write non-blocking code.

```js
async function getData() {
  try {
    const res = await fetch('api/data.json');
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error('Something went wrong', err);
  }
}
getData();
// Async/Await with Try...Catch
```

1. CALLBACKS

Callbacks are functions passed as arguments and executed later.

```js
function fetchUser(id, callback) {
  setTimeout(() => {
    callback({ id, name: 'John' });
  }, 1000);
}

fetchUser(1, (user) => {
  console.log(user);
});
```

PROS / CONS

- ✓ Simple and easy to understand.
- ✗ Callback Hell for multiple nested callbacks.

2. PROMISES

Promises represent a value that may be available now, later, or never.

```js
const promise = new Promise((resolve, reject) => {
  let success = true;
  setTimeout(() => {
    success ? resolve('Success!') : reject('Error!');
  }, 1000);
});

promise
  .then(res => console.log(res))
  .catch(err => console.error(err));
```

STATES

- **Pending** — Initial state.
- **Fulfilled** — Operation completed successfully.
- **Rejected** — Operation failed.

3. ASYNC / AWAIT

async await

Modern way to write async code that looks and reads like sync code.

```js
async function loadUser() {
  try {
    const res = await fetch('api/user.json');
    const user = await res.json();
    console.log(user);
  } catch (err) {
    console.error(err);
  }
}
loadUser();
```

WHY USE IT?

- ✓ Cleaner and more readable code.
- ✓ Easier error handling with try...catch.
- ✓ Avoids callback hell and .then() chains.

4. FETCH API (WITH ASYNC/AWAIT)

Use Fetch API to make HTTP requests.

```js
async function getPosts() {
  const res = await fetch('https://jsonplaceholder.typicode.com/posts');
  if (!res.ok) throw new Error('Network response was not ok');
  const posts = await res.json();
  console.log(posts);
}
getPosts();
```

COMMON METHODS

`fetch(url)`	Make a request
`res.json()`	Parse JSON response
`res.text()`	Parse text response
`res.ok`	Check if request was successful

QUICK TIPS

- ✓ Always use try...catch with async/await.
- ✓ Use async/await for cleaner code.
- ✓ Handle loading and error states in UI.
- ✓ Don't forget to return or await promises.

COMMON ERRORS

- ✗ Forgetting await => Returns a Promise, not the value
- ✗ Missing catch => Unhandled promise rejection
- ✗ Using async without try...catch => Harder to debug errors
- ✗ Mixing .then() and async/await => Leads to messy code

Save this post for quick reference!

Share with your dev friends!

Follow for more web dev content!

CODING ARTIST

09 / 12

09 ES6+ FEATURES

Modern JavaScript made simple.

```
// ES6+ makes JavaScript more powerful,
// cleaner and developer-friendly.
const name = 'ES6 Rules!';
const features = ['clean', 'modern', 'powerful'];
console.log(`${name} ${features[0]} & ${features[1]}!`);

// ES6+ = Better Syntax, Better Code ✨
```

1. LET & CONST

Block-scoped variables.

```
let name = 'Alice';
const age = 25;
// let -> can be reassigned
// const -> cannot be reassigned

if (true) {
  let x = 10;
  const y = 20;
}
// x and y are block-scoped
```

2. ARROW FUNCTIONS

Shorter syntax and lexical this.

```
const add = (a, b) => a + b;
const greet = name => `Hi, ${name}!`;

---

const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2);
```

No own this, arguments, or super.

3. TEMPLATE LITERALS

String interpolation made easy.

```
const user = 'John';
const msg = `Hello, ${user}!`
Today is ${new Date().toDateString()}`;
console.log(msg);
```

Use backticks (`) and ${ } to embed expressions.

4. DESTRUCTURING

Extract values from arrays/objects.

```
// Array
const [a, b, ...rest] = [1, 2, 3, 4];
// a = 1, b = 2, rest = [3, 4]

// Object
const { name, age } = { name: 'Sam', age: 30 };
// name = 'Sam', age = 30
```

Makes extracting values clean & easy.

5. SPREAD OPERATOR

Expand arrays, objects, strings.

```
// Array
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];
// [1, 2, 3, 4]

// Object
const obj = { a: 1, b: 2 };
const newObj = { ...obj, c: 3 };
// { a: 1, b: 2, c: 3 }
```

Great for copying, merging & passing data.

6. DEFAULT PARAMETERS

Set default values for function params.

```
const greet = (name = 'Guest') =>
  `Hello, ${name}!`;

greet();          // Hello, Guest!
greet('Alice');   // Hello, Alice!
```

Avoid undefined and write cleaner code.

7. REST PARAMETERS

Collect multiple arguments into an array.

```
const sum = (...nums) => {
  return nums.reduce((acc, n) => acc + n, 0);
};

sum(1, 2, 3, 4); // 10
```

Use ...before the parameter to gather the rest.

8. ENHANCED OBJECT LITERALS

Shorter syntax for objects.

```
const name = 'Tom';
const age = 28;
const obj = { name, age, sayHello() {
  return `Hi, I'm ${name}`;
}};
obj.sayHello(); // Hi, I'm Tom
```

No need to repeat keys!

9. MODULES (IMPORT / EXPORT)

Split code into reusable modules.

```
// math.js
export const add = (a, b) => a + b;
// app.js
import { add } from './math.js';
console.log(add(2, 3)); // 5
```

Use export to share, import to use.

BONUS FEATURES

- Classes – Cleaner OOP syntax.
- Promises – Better async handling.
- Map & Set – New data structures.
- Optional Chaining (?.) – Safe access.
- Nullish Coalescing (??) – Handle null/undefined.

```
// Optional Chaining
const user = { profile: { name: 'Alex' } };
console.log(user.profile?.name);   // Alex
console.log(user.address?.city);   // undefined
// Nullish Coalescing
const city = user.address?.city ?? 'Unknown City';
console.log(city);      // Unknown City
```

QUICK TIP

Adopt ES6+ features step by step and level up your JavaScript like a pro! 🚀

Save this post for quick reference!

Share with your dev friends!

Follow for more web dev content!

CODING ARTIST

10 / 12

10 ERROR HANDLING

Handle errors like a pro and make your code bulletproof.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Cannot divide by zero');
  }
  return a / b;
}
try {
  console.log(divide(10, 2));  // 5
  console.log(divide(10, 0));  // Error
} catch (err) {
  console.error('Oops! ' + err.message);
} finally {
  console.log('Execution completed');
}
```

THE TRY...CATCH...FINALLY BLOCK

1. try

try — Wrap the code that might throw an error.

```
try {
  // risky code
  let result = riskyOperation();
  console.log(result);
}
```

2. catch

catch — Catch the error and handle it gracefully.

```
} catch (error) {
  console.error('Error:',
    error.message);
  // handle the error
}
```

3. finally

finally — Always runs, whether there's an error or not.

```
finally {
  console.log('Clean up here');
  // close connections,
  // release resources, etc.
}
```

TYPES OF ERRORS

Syntax Errors
Mistakes in writing code that prevent it from running.

```
console.log('Hello'   // ✗
```

Missing closing parenthesis

Runtime Errors
Happen while the program is running.

```
let x = null;
console.log(x.length); // ✗
```

Cannot read properties of null

Logical Errors
Code runs without errors but gives the wrong result.

```
let total = 0;
for (let i = 0; i < 5; i++) {
  total += i; // ✗
}
```

Should be <= 5 to include 5

BUILT-IN ERROR TYPES

Error	Base class for all errors.
SyntaxError	Invalid syntax.
ReferenceError	Using undeclared variable.
TypeError	Wrong type of value.
RangeError	Value out of range.
URIError	Invalid URI.

CUSTOM ERRORS

Create your own error types for better control.

```
class ValidationError extends Error {
  construction(message) {
    super(message);
    this.name = 'ValidationError';
  }
}
function validateAge(age) {
  if (age < 18) throw new ValidationError('You must be 18+');
}
```

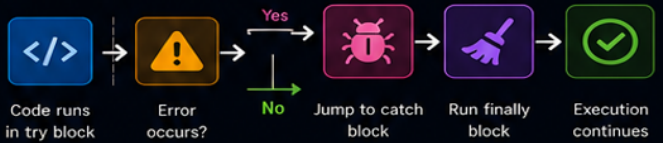
BEST PRACTICES

- Always use try...catch for code that can fail.
- Be specific in your error messages.
- Avoid empty catch blocks.
- Use finally for cleanup (close files, connections, timers).
- Create custom errors for better clarity.
- Don't swallow errors silently—log or rethrow them.
- Keep error handling simple and readable.

EXAMPLE: FETCH WITH ERROR HANDLING

```
async function getUser(id) {
  try {
    const res = await fetch('https://api.example.com/users/${id}');
    if (!res.ok) throw new Error('HTTP ${res.status}');
    const user = await res.json();
    return user;
  } catch (err) {
    console.error('Failed to fetch user:', err.message);
    return null;
  } finally {
    console.log('Request finished');
  }
}
```

FLOW OF ERROR HANDLING



QUICK TIPS

- Anticipate errors, don't just react to them.
- Log errors with context for easier debugging.
- A good error message saves hours of debugging.
- Robust error handling leads to reliable applications.

Save this post for quick reference!

Share with your dev friends!

Follow for more web dev content!

CODING ARTIST

11 / 12

11 WEB STORAGE

Store data in the browser with localStorage & sessionStorage.

```
// localStorage - data never expires
// sessionStorage - data cleared on tab close

localStorage.setItem('name', 'Coding Artist');
const name = localStorage.getItem('name');
localStorage.removeItem('name');
localStorage.clear();

sessionStorage.setItem('token', 'abc123');
const token = sessionStorage.getItem('token');
sessionStorage.removeItem('token');
sessionStorage.clear();
```

localStorage vs sessionStorage

Feature	localStorage	sessionStorage
Lifetime	Data is stored with no expiration.	Data exists only for the session (tab).
Scope	Shared across all tabs/windows from the same origin.	Isolated per tab/window.
Cleared When	Manually cleared or by user (browser data).	Tab/window is closed.
Use Cases	User preferences, themes, long-term data.	Form data, temporary tokens, wizard steps.
Storage Limit	~5MB per origin (varies by browser)	

STORAGE API METHODS

setItem(key, value)

Stores a key-value pair.

```
localStorage.setItem(
    'theme',
    'dark'
);
```

getItem(key)

Retrieves the value by key.

```
const theme =
localStorage.getItem(
  'theme');
```

removeItem(key)

Removes the key-value pair.

```
localStorage.removeItem(
  'theme');
```

clear()

Removes all stored data.

```
localStorage.clear();
```

key(index)

Gets the key at the given index.

```
const firstKey =
localStorage.key(0);
```

STORAGE EVENTS

The 'storage' event fires in all other tabs/windows of the same origin when localStorage changes.

```
window.addEventListener('storage', (e) => {
  console.log('Key:', e.key);
  console.log('Old Value:', e.oldValue);
  console.log('New Value:', e.newValue);
});
```

Note: The storage event does NOT fire in the same tab that made the change.

BEST PRACTICES

- Store only what you need.
- Always handle null when getting values.
- Use JSONstringify() for objects.
- Don't store sensitive data.
- Clear unnecessary data regularly.

STORE OBJECTS

Storage stores only strings. Use JSON methods.

```
// Save
const user = { name: 'Alex', age: 25 };
localStorage.setItem('user', JSON.stringify(user));

// Get
const data = JSON.parse(localStorage.getItem('user'));
console.log(data.name); // Alex
```

QUICK TIPS

- Use localStorage for long-term persistence.
- Use sessionStorage for temporary/session data.
- Great for improving UX without a database!

Save this post for quick reference!

Share with your dev friends!

Follow for more web dev content!

CODING ARTIST

12 / 12

12 DOM MANIPULATION

Change the content, structure & style of your web pages dynamically.

```js
// Select an element
const title = document.querySelector('#title');
// Change content
title.textContent = 'Hello, Coding Artist!';
// Change style
title.style.color = 'tomato';
title.style.fontSize = '2rem';
// Add a class
title.classList.add('highlight');
```

SELECTING ELEMENTS

getElementById()

Selects an element by its ID.

```js
const el = document.getElementById('id');
```

Returns a single element.

querySelector()

Selects the first element that matches a CSS selector.

```js
const el = document.querySelector('.box');
```

Great for modern websites.

querySelectorAll()

Selects all elements that match a CSS selector.

```js
const els = document.querySelectorAll('p');
```

Returns a NodeList.

getElementsByClassName()

Selects all elements by class name.

```js
const els = document.getElementsByClassName('box');
```

Returns an HTMLCollection.

MANIPULATING CONTENT & STRUCTURE

textContent

Sets or returns the text content.

```js
el.textContent = 'Hello!';
```

Pure text, no HTML.

innerHTML

Sets or returns the HTML content.

```js
el.innerHTML = '<b>Hi!</b>';
```

Ge careful with user input!

setAttribute()

Sets the value of an attribute.

```js
el.setAttribute('src', 'img.png');
```

Use for custom attributes too.

classList

Adds, removes, toggles CSS classes.

```js
el.classList.add('active');
el.classList.remove('hide');
el.classList.toggle('open');
```

Super useful!

createElement()

Creates a new HTML element.

```js
const div = document.createElement('div');
```

Element not added to DOM yet.

appendChild()

Adds a node as the last child of a parent.

```js
parent.appendChild(child);
```

Moves element if it exists.

prepend()

Adds a node as the first child of a parent.

```js
parent.prepend(child);
```

Great for inserting on top.

remove()

Removes an element from the DOM.

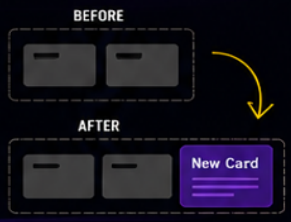
```js
el.remove();
```

Element is gone!

EXAMPLE: CREATE & ADD A CARD

```js
// Create a new card
const card = document.createElement('div');
card.className = 'card';
card.innerHTML = '<h3>New Card</h3><p>This card was added with JavaScript!</p>';

// Add to the container
const container = document.querySelector('#cards');
container.appendChild(card);
```



EVENT + DOM = POWER

```js
const btn = document.querySelector('#btn');
const msg = document.querySelector('#mssg');

btn.addEventListener('click', () => {
  msg.textContent = 'Button clicked!';
  msg.classList.add('show');
});
```

User clicks button → JS → DOM updates instantly!

BEST PRACTICES

- Select elements once and reuse them.
- Use textContent when inserting plain text.
- Avoid innerHTML with user input (XSS risk).
- Keep your DOM updates minimal & efficient.
- Use event delegation for better performance.

COMMON MISTAKES

- Forgetting that IDs must be unique.
- Using innerHTML unnecessarily.
- Not checking if an element exists.
- Adding too many DOM updates in loops.
- Not removing event listeners when needed.

QUICK TIPS

- Use DevTools to inspect elements.
- Practice by building small UI components.
- Learn by doing – experiment daily!
- DOM is your playground!

Save this post for quick reference!

Share with your dev friends!

Follow for more web dev content!