Complete JavaScript Cheatsheet - Beginner to Advanced

Table of Contents

Basics & Variables

Variable Declaration

```javascript
// var (function-scoped, can be redeclared)
var name = "John";
var age = 25;

// let (block-scoped, cannot be redeclared)
let email = "john@example.com";
let isActive = true;

// const (block-scoped, cannot be reassigned)
const PI = 3.14159;
const users = []; // Object contents can still be modified
```

Variable Naming Rules

- Must start with letter, underscore (_), or dollar sign ($)
- Can contain letters, numbers, underscores, dollar signs
- Case-sensitive
- Cannot use reserved keywords

Comments

```
// Single line comment

/*
Multi-line
comment
*/

/**
 * JSDoc comment for functions
 * @param {string} name - The name parameter
 * @returns {string} Greeting message
 */
```

Data Types

Primitive Types

```
// Number
let age = 25;
let price = 99.99;
let negative = -10;
let infinity = Infinity;
let notANumber = NaN;

// String
let firstName = "John";
let lastName = "Doe";
let template = `Hello ${firstName} ${lastName}`;

// Boolean
let isTrue = true;
let isFalse = false;

// Undefined
let undefinedVar;
console.log(undefinedVar); // undefined

// Null
let nullVar = null;

// Symbol (ES6)
let sym = Symbol("id");
let sym2 = Symbol("id");
console.log(sym === sym2); // false

// BigInt (ES2020)
let bigNumber = 1234567890123456789012345678901234567890n;
```

Type Checking

```
typeof 42; // "number"
typeof "hello"; // "string"
typeof true; // "boolean"
typeof undefined; // "undefined"
typeof null; // "object" (known quirk)
typeof {}; // "object"
typeof []; // "object"
typeof function () {}; // "function"

// Better type checking
Array.isArray([]); // true
Number.isInteger(42); // true
Number.isNaN(NaN); // true
```

Type Conversion

```
// Implicit conversion
"5" + 3; // "53" (string concatenation)
"5" - 3; // 2 (numeric subtraction)
"5" * 3; // 15
true + 1; // 2
false + 1; // 1

// Explicit conversion
String(123); // "123"
Number("123"); // 123
Number("123abc"); // NaN
Boolean(1); // true
Boolean(0); // false
parseInt("123px"); // 123
parseFloat("12.34px"); // 12.34
```

Operators

Arithmetic Operators

```
let a = 10,
  b = 3;

a + b; // 13 (addition)
a - b; // 7 (subtraction)
a * b; // 30 (multiplication)
a / b; // 3.333... (division)
```

```
a % b; // 1 (modulus/remainder)
a ** b; // 1000 (exponentiation - ES2016)

// Increment/Decrement
a++; // post-increment (returns old value, then increments)
++a; // pre-increment (increments first, then returns new value)
a--; // post-decrement
--a; // pre-decrement
```

Assignment Operators

```
let x = 10;

x += 5; // x = x + 5 (15)
x -= 3; // x = x - 3 (12)
x *= 2; // x = x * 2 (24)
x /= 4; // x = x / 4 (6)
x %= 5; // x = x % 5 (1)
x **= 3; // x = x ** 3 (1)
```

Comparison Operators

```
// Equality
5 == "5"; // true (loose equality, type coercion)
5 === "5"; // false (strict equality, no type coercion)
5 != "6"; // true (loose inequality)
5 !== "5"; // true (strict inequality)

// Relational
5 > 3; // true
5 < 3; // false
5 >= 5; // true
5 <= 4; // false
```

Logical Operators

```
// AND (&&)
true && true; // true
true && false; // false

// OR (||)
true || false; // true
false || false; // false

// NOT (!)
!true; // false
```

```
!false; // true

// Short-circuit evaluation
let user = null;
let name = user && user.name; // undefined (doesn't throw error)
let defaultName = name || "Guest"; // "Guest"

// Nullish coalescing (ES2020)
let value = null ?? "default"; // "default"
let value2 = 0 ?? "default"; // 0 (only null/undefined trigger default)
```

Bitwise Operators

```
let a = 5; // 101 in binary
let b = 3; // 011 in binary

a & b; // 1 (AND: 001)
a | b; // 7 (OR: 111)
a ^ b; // 6 (XOR: 110)
~a; // -6 (NOT: invert all bits)
a << 1; // 10 (left shift: 1010)
a >> 1; // 2 (right shift: 10)
a >>> 1; // 2 (unsigned right shift)
```

Control Structures

Conditional Statements

```
// if...else
let age = 18;
if (age >= 18) {
  console.log("Adult");
} else if (age >= 13) {
  console.log("Teenager");
} else {
  console.log("Child");
}

// Ternary operator
let status = age >= 18 ? "Adult" : "Minor";

// Switch statement
let day = "Monday";
switch (day) {
  case "Monday":
  case "Tuesday":
    console.log("Beginning of week");
    break;
  case "Friday":
    console.log("TGIF!");
    break;
  default:
    console.log("Regular day");
}
```

Loops

```
// for loop
for (let i = 0; i < 5; i++) {
  console.log(i); // 0, 1, 2, 3, 4
}

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

// do...while loop
let j = 0;
do {
  console.log(j);
  j++;
} while (j < 5);

// for...in loop (for object properties)
let obj = { a: 1, b: 2, c: 3 };
for (let key in obj) {
  console.log(key, obj[key]); // a 1, b 2, c 3
}

// for...of loop (for iterable values)
let arr = [1, 2, 3];
for (let value of arr) {
  console.log(value); // 1, 2, 3
}

// Loop control
for (let i = 0; i < 10; i++) {
  if (i === 3) continue; // skip iteration
  if (i === 7) break; // exit loop
  console.log(i);
}
```

Functions

Function Declarations

```javascript
// Function declaration (hoisted)
function greet(name) {
  return `Hello, ${name}!`;
}

// Function expression (not hoisted)
const greet2 = function (name) {
  return `Hello, ${name}!`;
};

// Arrow functions (ES6)
const greet3 = (name) => {
  return `Hello, ${name}!`;
};

// Concise arrow function
const greet4 = (name) => `Hello, ${name}!`;
const add = (a, b) => a + b;
const square = (x) => x * x;
```

Function Parameters

```javascript
// Default parameters
function greet(name = "World") {
  return `Hello, ${name}!`;
}

// Rest parameters
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
sum(1, 2, 3, 4); // 10

// Destructuring parameters
function createUser({ name, age, email }) {
  return { name, age, email, id: Date.now() };
}
createUser({ name: "John", age: 25, email: "john@example.com" });
```

Function Scope and Closures

```javascript
// Function scope
function outer() {
  let outerVar = "I'm outside!";

  function inner() {
    let innerVar = "I'm inside!";
    console.log(outerVar); // Can access outer variable
  }

  inner();
  // console.log(innerVar); // Error: innerVar is not defined
}

// Closures
function createCounter() {
  let count = 0;
  return function () {
    return ++count;
  };
}
const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Higher-Order Functions

```
// Function that takes another function as parameter
function operation(a, b, callback) {
  return callback(a, b);
}

const result = operation(5, 3, (x, y) => x + y); // 8

// Function that returns a function
function multiplier(factor) {
  return function (number) {
    return number * factor;
  };
}
const double = multiplier(2);
console.log(double(5)); // 10
```

Objects

Object Creation

```
// Object literal
let person = {
  name: "John",
  age: 30,
  city: "New York",
};
```

```javascript
// Object constructor
let person2 = new Object();
person2.name = "Jane";
person2.age = 25;

// Constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function () {
    return `Hello, I'm ${this.name}`;
  };
}
let person3 = new Person("Bob", 35);
```

Object Properties and Methods

```javascript
let obj = {
  property: "value",
  method: function () {
    return "I'm a method";
  },
  // ES6 method shorthand
  shortMethod() {
    return "I'm a short method";
  },
};

// Accessing properties
console.log(obj.property); // "value"
console.log(obj["property"]); // "value"

// Dynamic property access
let prop = "property";
console.log(obj[prop]); // "value"

// Adding/modifying properties
obj.newProperty = "new value";
obj["another"] = "another value";

// Deleting properties
delete obj.property;
```

Object Methods

```javascript
let person = { name: "John", age: 30, city: "NYC" };

// Object.keys() - returns array of keys
Object.keys(person); // ["name", "age", "city"]
```

```
// Object.values() - returns array of values
Object.values(person); // ["John", 30, "NYC"]

// Object.entries() - returns array of [key, value] pairs
Object.entries(person); // [["name", "John"], ["age", 30], ["city", "NYC"]]

// Object.assign() - copy properties
let copy = Object.assign({}, person);
let merged = Object.assign({}, person, { country: "USA" });

// Object.freeze() - prevent modifications
Object.freeze(person);
person.age = 31; // Won't work

// Object.seal() - prevent adding/removing properties
Object.seal(person);

// hasOwnProperty()
person.hasOwnProperty("name"); // true
```

Destructuring Objects

```
let person = { name: "John", age: 30, city: "NYC" };

// Basic destructuring
let { name, age } = person;
console.log(name, age); // "John", 30

// Renaming variables
let { name: fullName, age: years } = person;

// Default values
let { name, age, country = "USA" } = person;

// Nested destructuring
let user = {
  id: 1,
  profile: {
    name: "John",
    email: "john@example.com",
  },
};
let {
  profile: { name, email },
} = user;
```

Arrays

Array Creation

```
// Array literal
let fruits = ["apple", "banana", "orange"];

// Array constructor
let numbers = new Array(1, 2, 3, 4, 5);
let empty = new Array(5); // Creates array with 5 empty slots

// Array.from()
let chars = Array.from("hello"); // ["h", "e", "l", "l", "o"]
let range = Array.from({ length: 5 }, (_, i) => i); // [0, 1, 2, 3, 4]

// Array.of()
let nums = Array.of(1, 2, 3); // [1, 2, 3]
```

Array Methods - Mutating

```
let arr = [1, 2, 3];

// push() - add to end
arr.push(4, 5); // [1, 2, 3, 4, 5]

// pop() - remove from end
let last = arr.pop(); // last = 5, arr = [1, 2, 3, 4]

// unshift() - add to beginning
arr.unshift(0); // [0, 1, 2, 3, 4]

// shift() - remove from beginning
let first = arr.shift(); // first = 0, arr = [1, 2, 3, 4]

// splice() - add/remove at any position
arr.splice(1, 2, "a", "b"); // Remove 2 elements at index 1, add 'a', 'b'
// arr = [1, 'a', 'b', 4]

// sort() - sort array
let names = ["Charlie", "Alice", "Bob"];
names.sort(); // ["Alice", "Bob", "Charlie"]

// Custom sort
let numbers = [10, 5, 40, 25, 1000, 1];
numbers.sort((a, b) => a - b); // [1, 5, 10, 25, 40, 1000]

// reverse() - reverse array
arr.reverse(); // ['b', 'a', 1, 4]
```

Array Methods - Non-Mutating

```
let numbers = [1, 2, 3, 4, 5];

// slice() - extract portion
let portion = numbers.slice(1, 4); // [2, 3, 4]

// concat() - join arrays
let more = numbers.concat([6, 7, 8]); // [1, 2, 3, 4, 5, 6, 7, 8]

// join() - convert to string
let str = numbers.join("-"); // "1-2-3-4-5"

// indexOf() / lastIndexOf() - find index
numbers.indexOf(3); // 2
numbers.lastIndexOf(3); // 2

// includes() - check if element exists
numbers.includes(3); // true
```

Array Iteration Methods

```
let numbers = [1, 2, 3, 4, 5];

// forEach() - execute function for each element
numbers.forEach((num, index) => {
  console.log(`Index ${index}: ${num}`);
});

// map() - transform each element
let doubled = numbers.map((num) => num * 2); // [2, 4, 6, 8, 10]

// filter() - filter elements
let evens = numbers.filter((num) => num % 2 === 0); // [2, 4]

// find() - find first matching element
let found = numbers.find((num) => num > 3); // 4

// findIndex() - find index of first matching element
let index = numbers.findIndex((num) => num > 3); // 3

// some() - test if any element passes test
let hasEven = numbers.some((num) => num % 2 === 0); // true

// every() - test if all elements pass test
let allPositive = numbers.every((num) => num > 0); // true

// reduce() - reduce to single value
let sum = numbers.reduce((acc, num) => acc + num, 0); // 15
let max = numbers.reduce((max, num) => (num > max ? num : max)); // 5
```

Array Destructuring

```
let arr = [1, 2, 3, 4, 5];

// Basic destructuring
let [first, second] = arr; // first = 1, second = 2

// Skip elements
let [, , third] = arr; // third = 3

// Rest elements
let [head, ...tail] = arr; // head = 1, tail = [2, 3, 4, 5]

// Default values
let [a, b, c, d, e, f = 0] = arr; // f = 0

// Swapping variables
let x = 1,
  y = 2;
[x, y] = [y, x]; // x = 2, y = 1
```

Strings

String Creation

```
let str1 = "Hello World";
let str2 = "Hello World";
let str3 = `Hello World`; // Template literal

// Escape characters
let escaped = 'He said "Hello" to me';
let newline = "Line 1\nLine 2";
let tab = "Column1\tColumn2";
```

Template Literals

```
let name = "John";
let age = 30;

// String interpolation
let message = `Hello, my name is ${name} and I'm ${age} years old.`;

// Multi-line strings
let multiline = `
    This is line 1
    This is line 2
    This is line 3
`;

// Expression evaluation
let calculation = `The sum of 5 + 3 is ${5 + 3}`;
```

String Methods

```
let str = "Hello World";

// Length
str.length; // 11

// Case conversion
str.toUpperCase(); // "HELLO WORLD"
str.toLowerCase(); // "hello world"

// Substring extraction
str.charAt(0); // "H"
str.charCodeAt(0); // 72 (ASCII code)
str.substring(0, 5); // "Hello"
str.substr(6, 5); // "World"
str.slice(0, 5); // "Hello"
str.slice(-5); // "World"

// Search methods
str.indexOf("o"); // 4 (first occurrence)
str.lastIndexOf("o"); // 7 (last occurrence)
str.includes("World"); // true
str.startsWith("Hello"); // true
str.endsWith("World"); // true

// Replace
str.replace("World", "JavaScript"); // "Hello JavaScript"
str.replace(/o/g, "0"); // "Hell0 W0rld" (regex replace all)

// Split
str.split(" "); // ["Hello", "World"]
str.split(""); // ["H", "e", "l", "l", "o", " ", "W", "o", "r", "l", "d"]

// Trim
let padded = "  Hello World  ";
padded.trim(); // "Hello World"
padded.trimStart(); // "Hello World  "
padded.trimEnd(); // "  Hello World"

// Repeat
"Hi".repeat(3); // "HiHiHi"

// Padding (ES2017)
```

```
"5".padStart(3, "0"); // "005"
"5".padEnd(3, "0"); // "500"
```

DOM Manipulation

Selecting Elements

```
// By ID
let element = document.getElementById("myId");

// By class name
let elements = document.getElementsByClassName("myClass");

// By tag name
let divs = document.getElementsByTagName("div");

// Query selector (CSS selectors)
let first = document.querySelector(".myClass");
let all = document.querySelectorAll(".myClass");

// Modern methods
let element2 = document.querySelector("#myId");
let elements2 = document.querySelectorAll("div.myClass");
```

Modifying Elements

```
let element = document.querySelector("#myElement");

// Content
element.innerHTML = "<strong>Bold text</strong>";
element.textContent = "Plain text";
element.innerText = "Visible text only";

// Attributes
element.setAttribute("class", "newClass");
element.getAttribute("class");
element.removeAttribute("class");
element.hasAttribute("class");

// Properties
element.id = "newId";
element.className = "newClass";
element.classList.add("class1");
element.classList.remove("class2");
element.classList.toggle("active");
element.classList.contains("active");

// Styles
```

```
element.style.color = "red";
element.style.backgroundColor = "blue";
element.style.fontSize = "16px";
```

Creating and Manipulating Elements

```
// Create element
let newDiv = document.createElement("div");
newDiv.textContent = "New content";
newDiv.className = "myClass";

// Append to parent
let parent = document.querySelector("#parent");
parent.appendChild(newDiv);
parent.append(newDiv); // Can append multiple nodes/strings
parent.prepend(newDiv); // Add to beginning

// Insert at specific position
parent.insertBefore(newDiv, parent.firstChild);

// Replace element
parent.replaceChild(newDiv, oldElement);

// Remove element
parent.removeChild(element);
element.remove(); // Modern method

// Clone element
let clone = element.cloneNode(true); // true for deep clone
```

Events

Event Listeners

```
let button = document.querySelector("#myButton");

// Add event listener
button.addEventListener("click", function (event) {
  console.log("Button clicked!");
  console.log(event); // Event object
});

// Arrow function event listener
button.addEventListener("click", (e) => {
  e.preventDefault(); // Prevent default behavior
  e.stopPropagation(); // Stop event bubbling
});
```

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

// Event listener options
button.addEventListener("click", handleClick, {
  once: true, // Execute only once
  passive: true, // Never calls preventDefault
  capture: true, // Capture phase instead of bubble
});
```

Common Events

```
// Mouse events
element.addEventListener("click", handler);
element.addEventListener("dblclick", handler);
element.addEventListener("mousedown", handler);
element.addEventListener("mouseup", handler);
element.addEventListener("mouseover", handler);
element.addEventListener("mouseout", handler);
element.addEventListener("mousemove", handler);

// Keyboard events
document.addEventListener("keydown", (e) => {
  console.log(`Key pressed: ${e.key}`);
  console.log(`Key code: ${e.keyCode}`);
  console.log(`Ctrl pressed: ${e.ctrlKey}`);
  console.log(`Shift pressed: ${e.shiftKey}`);
});

// Form events
form.addEventListener("submit", (e) => {
  e.preventDefault();
  // Handle form submission
});

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

input.addEventListener("change", handler);
input.addEventListener("focus", handler);
input.addEventListener("blur", handler);

// Window events
window.addEventListener("load", () => {
  console.log("Page fully loaded");
});
```

```
window.addEventListener("resize", () => {
  console.log(`Window size: ${window.innerWidth}x${window.innerHeight}`);
});

window.addEventListener("scroll", () => {
  console.log(`Scroll position: ${window.scrollY}`);
});
```

Event Delegation

```
// Instead of adding listeners to many elements
document.querySelector("#parent").addEventListener("click", (e) => {
  if (e.target.matches(".child-button")) {
    console.log("Child button clicked!");
  }
});
```

Asynchronous JavaScript

Callbacks

```
// Simple callback
function fetchData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
}

fetchData((data) => {
  console.log(data); // "Data received" after 1 second
});

// Callback hell example
getData((a) => {
  getMoreData(a, (b) => {
    getEvenMoreData(b, (c) => {
      // This nesting can get very deep
    });
  });
});
```

Promises

```javascript
// Creating a promise
let promise = new Promise((resolve, reject) => {
  let success = true;

  setTimeout(() => {
    if (success) {
      resolve("Operation successful!");
    } else {
      reject("Operation failed!");
    }
  }, 1000);
});

// Using promises
promise
  .then((result) => {
    console.log(result);
    return "Next step";
  })
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  })
  .finally(() => {
    console.log("Promise completed");
  });

// Promise methods
Promise.all([promise1, promise2, promise3]).then((results) => {
  // All promises resolved
  console.log(results);
});

Promise.allSettled([promise1, promise2, promise3]).then((results) => {
  // All promises settled (resolved or rejected)
  results.forEach((result) => {
    if (result.status === "fulfilled") {
      console.log(result.value);
    } else {
      console.log(result.reason);
    }
  });
});

Promise.race([promise1, promise2, promise3]).then((result) => {
  // First promise to resolve/reject
  console.log(result);
});
```

Async/Await

```
// Async function
async function fetchData() {
  try {
    let response = await fetch("https://api.example.com/data");
    let data = await response.json();
    return data;
  } catch (error) {
    console.error("Error:", error);
    throw error;
  }
}

// Using async function
fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.error(error));

// Or with async/await
async function main() {
  try {
    let data = await fetchData();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

// Parallel execution
async function fetchMultiple() {
  try {
    // Sequential (slow)
    let result1 = await fetch("/api/data1");
    let result2 = await fetch("/api/data2");

    // Parallel (fast)
    let [result3, result4] = await Promise.all([
      fetch("/api/data3"),
      fetch("/api/data4"),
    ]);

    return { result1, result2, result3, result4 };
  } catch (error) {
    console.error(error);
  }
}
```

Fetch API

```
// GET request
fetch("https://api.example.com/users")
```

```
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));

// POST request
fetch("https://api.example.com/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    Authorization: "Bearer token123",
  },
  body: JSON.stringify({
    name: "John Doe",
    email: "john@example.com",
  }),
})
  .then((response) => response.json())
  .then((data) => console.log(data));

// With async/await
async function createUser(userData) {
  try {
    const response = await fetch("/api/users", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(userData),
    });

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const data = await response.json();
    return data;
  } catch (error) {
    console.error("Error creating user:", error);
    throw error;
  }
}
```

ES6+ Features

Let and Const

```
// Block scoping
if (true) {
  let blockScoped = "I'm block scoped";
  const alsoBlockScoped = "Me too";
  var functionScoped = "I'm function scoped";
}
// console.log(blockScoped); // Error
// console.log(alsoBlockScoped); // Error
console.log(functionScoped); // Works

// Temporal dead zone
console.log(hoisted); // undefined (var hoisting)
// console.log(notHoisted); // Error (let/const not hoisted)
var hoisted = "I'm hoisted";
let notHoisted = "I'm not hoisted";
```

Arrow Functions

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

// Arrow function variations
const add1 = (a, b) => a + b;
const add2 = (a, b) => {
  return a + b;
};
const square = (x) => x * x;
const greet = () => "Hello!";

// Arrow functions and 'this'
const obj = {
  name: "John",

  // Regular function - 'this' refers to obj
  regularMethod: function () {
    console.log(this.name); // "John"
  },

  // Arrow function - 'this' refers to outer scope
  arrowMethod: () => {
    console.log(this.name); // undefined (or global)
  },
};
```

Template Literals

```
const name = "John";
const age = 30;

// String interpolation
const message = `Hello, ${name}! You are ${age} years old.`;

// Multi-line strings
const html = `
    <div class="user">
        <h2>${name}</h2>
        <p>Age: ${age}</p>
    </div>
`;

// Tagged template literals
function highlight(strings, ...values) {
  return strings.reduce((result, string, i) => {
    const value = values[i] ? `<mark>${values[i]}</mark>` : "";
    return result + string + value;
  }, "");
}

const highlighted = highlight`Hello ${name}, you are ${age} years old!`;
```

Destructuring Assignment

```
// Array destructuring
const colors = ["red", "green", "blue"];
const [primary, secondary, tertiary] = colors;
const [first, , third] = colors; // Skip middle element
const [head, ...rest] = colors; // Rest elements

// Object destructuring
const person = { name: "John", age: 30, city: "NYC" };
const { name, age } = person;
const { name: fullName, age: years } = person; // Rename
const { name, age, country = "USA" } = person; // Default values

// Nested destructuring
const user = {
  profile: {
    name: "John",
    contact: {
      email: "john@example.com",
    },
  },
};
const {
  profile: {
    name,
    contact: { email },
  },
} = user;

// Function parameter destructuring
function greetUser({ name, age = 0 }) {
  return `Hello ${name}, you are ${age} years old`;
}
greetUser({ name: "John", age: 30 });
```

Spread and Rest Operators

```
// Spread operator (...)
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

const original = [1, 2, 3];
const copy = [...original]; // Shallow copy

// Object spread
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const merged = { ...obj1, ...obj2 }; // { a: 1, b: 2, c: 3, d: 4 }

// Function arguments
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
sum(1, 2, 3, 4, 5); // 15

const numbers = [1, 2, 3, 4, 5];
console.log(Math.max(...numbers)); // 5
```

Enhanced Object Literals

```
const name = "John";
const age = 30;

// Property shorthand
const person = { name, age }; // Instead of { name: name, age: age }

// Method shorthand
const calculator = {
  add(a, b) {
    return a + b;
  }, // Instead of add: function(a, b) { ... }
  subtract(a, b) {
    return a - b;
```

```javascript
  },
};

// Computed property names
const prop = "dynamicProperty";
const obj = {
  [prop]: "value",
  ["method" + "Name"]() {
    return "dynamic method";
  },
};
```

Classes

```javascript
// Class declaration
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  // Instance method
  greet() {
    return `Hello, I'm ${this.name}`;
  }

  // Static method
  static species() {
    return "Homo sapiens";
  }

  // Getter
  get info() {
    return `${this.name} is ${this.age} years old`;
  }

  // Setter
  set age(value) {
    if (value < 0) throw new Error("Age cannot be negative");
    this._age = value;
  }

  get age() {
    return this._age;
  }
}

// Inheritance
class Student extends Person {
  constructor(name, age, school) {
    super(name, age); // Call parent constructor
    this.school = school;
  }

  greet() {
    return `${super.greet()} and I study at ${this.school}`;
  }

  // Private fields (ES2020)
  #privateField = "secret";

  #privateMethod() {
    return this.#privateField;
  }
}

const student = new Student("John", 20, "MIT");
console.log(student.greet());
console.log(Person.species());
```

Default Parameters

```
function greet(name = "World", punctuation = "!") {
  return `Hello, ${name}${punctuation}`;
}

greet(); // "Hello, World!"
greet("John"); // "Hello, John!"
greet("John", "?"); // "Hello, John?"

// With destructuring
function createUser({ name = "Anonymous", age = 0, active = true } = {}) {
  return { name, age, active };
}

createUser(); // { name: 'Anonymous', age: 0, active: true }
createUser({ name: "John" }); // { name: 'John', age: 0, active: true }
```

Modules (ES6)

```
// Export (in module.js)
export const PI = 3.14159;
export function circle(radius) {
  return PI * radius * radius;
}

export default class Calculator {
  add(a, b) {
    return a + b;
  }
  subtract(a, b) {
    return a - b;
  }
}

// Named exports
export { PI, circle };

// Import (in main.js)
import Calculator, { PI, circle } from "./module.js";
import { PI as pi } from "./module.js"; // Rename import
import * as MathUtils from "./module.js"; // Import all

// Dynamic imports
async function loadModule() {
  const module = await import("./module.js");
  return module.default; // Default export
}
```

Iterators and Generators

```
// Iterator
const iterable = {
  data: [1, 2, 3, 4, 5],
  [Symbol.iterator]() {
    let index = 0;
    return {
      next: () => {
        if (index < this.data.length) {
          return { value: this.data[index++], done: false };
        } else {
          return { done: true };
        }
      },
    };
  },
};

// Generator function
function* numberGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

function* infiniteNumbers() {
  let num = 0;
  while (true) {
    yield num++;
  }
}
```

```javascript
// Using generators
const gen = numberGenerator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }

// Generator with parameters
function* fibonacci() {
  let a = 0,
    b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

const fib = fibonacci();
console.log(fib.next().value); // 0
console.log(fib.next().value); // 1
console.log(fib.next().value); // 1
```

Symbol

```javascript
// Creating symbols
const sym1 = Symbol();
const sym2 = Symbol("description");
const sym3 = Symbol("description");

console.log(sym2 === sym3); // false (each symbol is unique)

// Using symbols as object properties
const id = Symbol("id");
const user = {
  name: "John",
  [id]: 12345,
};

console.log(user[id]); // 12345
console.log(Object.keys(user)); // ['name'] (symbol properties are hidden)

// Well-known symbols
const obj = {
  [Symbol.iterator]: function* () {
    yield 1;
    yield 2;
    yield 3;
  },
};

for (const value of obj) {
  console.log(value); // 1, 2, 3
}
```

Advanced Concepts

Closures

```
// Basic closure
function outerFunction(x) {
  return function innerFunction(y) {
    return x + y;
  };
}

const addFive = outerFunction(5);
console.log(addFive(3)); // 8

// Module pattern using closures
const Counter = (function () {
  let count = 0;

  return {
    increment: function () {
      count++;
    },
    decrement: function () {
      count--;
    },
    getCount: function () {
      return count;
    },
  };
})();

Counter.increment();
console.log(Counter.getCount()); // 1

// Function factory
function createMultiplier(multiplier) {
  return function (x) {
    return x * multiplier;
  };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);
console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Prototypes and Inheritance

```javascript
// Constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Adding methods to prototype
Person.prototype.greet = function () {
  return `Hello, I'm ${this.name}`;
};

Person.prototype.getAge = function () {
  return this.age;
};

// Creating instances
const person1 = new Person("John", 30);
const person2 = new Person("Jane", 25);

console.log(person1.greet()); // "Hello, I'm John"

// Inheritance with prototypes
function Student(name, age, school) {
  Person.call(this, name, age); // Call parent constructor
  this.school = school;
}

// Set up inheritance
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

// Add Student-specific methods
Student.prototype.study = function () {
  return `${this.name} is studying at ${this.school}`;
};

const student = new Student("Bob", 20, "MIT");
console.log(student.greet()); // "Hello, I'm Bob"
console.log(student.study()); // "Bob is studying at MIT"

// Prototype chain methods
console.log(person1.hasOwnProperty("name")); // true
console.log(person1.hasOwnProperty("greet")); // false
console.log("greet" in person1); // true
console.log(Object.getPrototypeOf(person1) === Person.prototype); // true
```

'this' Keyword

```
// Global context
console.log(this); // Window object (in browser) or global (in Node.js)

// Object method
const obj = {
  name: "John",
  greet: function () {
    console.log(this.name); // 'John'
  },
};

// Function context
function regularFunction() {
  console.log(this); // Window object (non-strict mode) or undefined (strict mode)
}

// Arrow function - inherits 'this' from enclosing scope
const arrowFunction = () => {
  console.log(this); // Same as outer scope
};

// Constructor function
function Person(name) {
  this.name = name; // 'this' refers to new instance
}

// Explicit binding
const person = { name: "John" };
function sayName() {
  console.log(this.name);
}

sayName.call(person); // 'John'
sayName.apply(person); // 'John'
const boundFunction = sayName.bind(person);
boundFunction(); // 'John'

// Event handlers
button.addEventListener("click", function () {
  console.log(this); // The button element
});

button.addEventListener("click", () => {
  console.log(this); // Outer scope (not the button)
});
```

Call, Apply, and Bind

```
function introduce(greeting, punctuation) {
  return `${greeting}, I'm ${this.name}${punctuation}`;
}

const person = { name: "John" };

// call() - passes arguments individually
console.log(introduce.call(person, "Hello", "!")); // "Hello, I'm John!"

// apply() - passes arguments as array
console.log(introduce.apply(person, ["Hi", "."])); // "Hi, I'm John."

// bind() - returns new function with bound context
const boundIntroduce = introduce.bind(person);
console.log(boundIntroduce("Hey", "?")); // "Hey, I'm John?"

// Partial application with bind
const greetJohn = introduce.bind(person, "Hello");
console.log(greetJohn("!")); // "Hello, I'm John!"

// Using call with array methods
const numbers = [1, 2, 3, 4, 5];
const max = Math.max.apply(null, numbers); // or Math.max(...numbers)
console.log(max); // 5

// Borrowing methods
const arrayLike = { 0: "a", 1: "b", 2: "c", length: 3 };
const array = Array.prototype.slice.call(arrayLike);
console.log(array); // ['a', 'b', 'c']
```

Higher-Order Functions

```
// Function that takes function as parameter
function withLogging(fn) {
  return function (...args) {
    console.log(`Calling function with args: ${args}`);
    const result = fn(...args);
    console.log(`Function returned: ${result}`);
    return result;
  };
}

const add = (a, b) => a + b;
const loggedAdd = withLogging(add);
loggedAdd(2, 3); // Logs function call and result

// Function composition
const compose = (f, g) => (x) => f(g(x));
const addOne = (x) => x + 1;
const multiplyByTwo = (x) => x * 2;
const addOneThenMultiply = compose(multiplyByTwo, addOne);
console.log(addOneThenMultiply(3)); // 8 ((3 + 1) * 2)
```

```
// Currying
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args);
    } else {
      return function (...nextArgs) {
        return curried(...args, ...nextArgs);
      };
    }
  };
}

const multiply = (a, b, c) => a * b * c;
const curriedMultiply = curry(multiply);
console.log(curriedMultiply(2)(3)(4)); // 24
console.log(curriedMultiply(2, 3)(4)); // 24
```

Event Loop and Asynchronous Behavior

```
// Understanding the event loop
console.log("1");

setTimeout(() => console.log("2"), 0);

Promise.resolve().then(() => console.log("3"));

console.log("4");

// Output: 1, 4, 3, 2
// Explanation:
// - Synchronous code runs first (1, 4)
// - Microtasks (Promises) run before macrotasks (setTimeout)
// - So Promise callback (3) runs before setTimeout callback (2)

// Microtasks vs Macrotasks
console.log("Start");

// Macrotask
setTimeout(() => console.log("setTimeout 1"), 0);
setTimeout(() => console.log("setTimeout 2"), 0);

// Microtask
Promise.resolve().then(() => console.log("Promise 1"));
Promise.resolve().then(() => console.log("Promise 2"));

console.log("End");

// Output: Start, End, Promise 1, Promise 2, setTimeout 1, setTimeout 2
```

Memory Management and Garbage Collection

```javascript
// Memory leaks to avoid

// 1. Global variables
var globalLeak = "This stays in memory";

// 2. Forgotten timers
const interval = setInterval(() => {
  // Do something
}, 1000);
// clearInterval(interval); // Don't forget to clear!

// 3. Closures holding references
function createHandler() {
  const largeData = new Array(1000000).fill("data");

  return function () {
    // If this closure is kept alive, largeData won't be garbage collected
    console.log("Handler called");
  };
}

// 4. Detached DOM nodes
let button = document.querySelector("#myButton");
document.body.removeChild(button);
// button still holds reference to DOM node

// Good practices
function properCleanup() {
  const element = document.querySelector("#temp");
  const handler = () => console.log("clicked");

  element.addEventListener("click", handler);

  // Cleanup function
  return function cleanup() {
    element.removeEventListener("click", handler);
    element = null; // Remove reference
  };
}

// Using WeakMap and WeakSet for weak references
const weakData = new WeakMap();
const obj = {};
weakData.set(obj, "some data");
// When obj is garbage collected, the WeakMap entry is automatically removed
```

Error Handling

Try-Catch-Finally

```
// Basic try-catch
try {
  let result = riskyOperation();
  console.log(result);
} catch (error) {
  console.error("An error occurred:", error.message);
} finally {
  console.log("This always runs");
}

// Specific error handling
try {
  JSON.parse("invalid json");
} catch (error) {
  if (error instanceof SyntaxError) {
    console.log("JSON syntax error");
  } else if (error instanceof ReferenceError) {
    console.log("Reference error");
  } else {
    console.log("Unknown error:", error);
  }
}

// Nested try-catch
try {
  try {
    throw new Error("Inner error");
  } catch (innerError) {
    console.log("Caught inner error");
    throw new Error("Outer error");
  }
} catch (outerError) {
  console.log("Caught outer error:", outerError.message);
}
```

Custom Errors

```
// Custom error class
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = "ValidationError";
    this.field = field;
  }
}

class NetworkError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.name = "NetworkError";
    this.statusCode = statusCode;
  }
}

// Using custom errors
function validateUser(user) {
  if (!user.name) {
    throw new ValidationError("Name is required", "name");
  }
  if (!user.email) {
    throw new ValidationError("Email is required", "email");
  }
}

try {
  validateUser({ name: "John" });
} catch (error) {
  if (error instanceof ValidationError) {
    console.log(`Validation failed for ${error.field}: ${error.message}`);
  }
}
```

Error Handling with Promises

```
// Promise error handling
fetch("/api/data")
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP ${response.status}: ${response.statusText}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.error("Fetch error:", error.message);
  });

// Async/await error handling
async function fetchUserData(userId) {
  try {
    const response = await fetch(`/api/users/${userId}`);

    if (!response.ok) {
      throw new NetworkError(
        `Failed to fetch user: ${response.statusText}`,
        response.status
      );
    }

    const userData = await response.json();
    return userData;
  } catch (error) {
    if (error instanceof NetworkError) {
      console.error(`Network error (${error.statusCode}): ${error.message}`);
      // Handle network-specific errors
    } else {
      console.error("Unexpected error:", error);
      // Handle other errors
    }
    throw error; // Re-throw if needed
  }
}

// Multiple async operations with error handling
async function processMultipleUsers(userIds) {
  const results = [];
  const errors = [];

  for (const userId of userIds) {
    try {
      const userData = await fetchUserData(userId);
      results.push(userData);
    } catch (error) {
      errors.push({ userId, error: error.message });
    }
  }

  return { results, errors };
}
```

Global Error Handling

```
// Unhandled promise rejections
window.addEventListener("unhandledrejection", (event) => {
  console.error("Unhandled promise rejection:", event.reason);
  event.preventDefault(); // Prevent default browser behavior
});

// Global error handler
window.addEventListener("error", (event) => {
  console.error("Global error:", {
    message: event.message,
    filename: event.filename,
    lineno: event.lineno,
    colno: event.colno,
    error: event.error,
  });
});
```

Best Practices

Code Organization

```javascript
// Use meaningful variable names
// Bad
const d = new Date();
const u = users.filter((u) => u.a);

// Good
const currentDate = new Date();
const activeUsers = users.filter((user) => user.isActive);

// Use constants for magic numbers
// Bad
if (user.age >= 18) {
  /* ... */
}

// Good
const LEGAL_AGE = 18;
if (user.age >= LEGAL_AGE) {
  /* ... */
}

// Function should do one thing
// Bad
function processUserAndSendEmail(user) {
  // Validate user
  if (!user.name) throw new Error("Name required");

  // Save to database
  database.save(user);

  // Send email
  emailService.send(user.email, "Welcome!");
}

// Good
function validateUser(user) {
  if (!user.name) throw new Error("Name required");
}

function saveUser(user) {
  return database.save(user);
}

function sendWelcomeEmail(user) {
  return emailService.send(user.email, "Welcome!");
}
```

```
async function processUser(user) {
  validateUser(user);
  await saveUser(user);
  await sendWelcomeEmail(user);
}
```

Performance Best Practices

```
// Use const and let instead of var
const API_URL = "https://api.example.com";
let counter = 0;

// Avoid global variables
(function () {
  // Your code here
})();

// Use strict mode
("use strict");

// Efficient DOM manipulation
// Bad - causes multiple reflows
for (let i = 0; i < 1000; i++) {
  document.body.appendChild(document.createElement("div"));
}

// Good - batch DOM updates
const fragment = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
  fragment.appendChild(document.createElement("div"));
}
document.body.appendChild(fragment);

// Debounce expensive operations
function debounce(func, wait) {
  let timeout;
  return function executedFunction(...args) {
    const later = () => {
      clearTimeout(timeout);
      func(...args);
    };
    clearTimeout(timeout);
    timeout = setTimeout(later, wait);
  };
}

const debouncedSearch = debounce(searchFunction, 300);
searchInput.addEventListener("input", debouncedSearch);

// Use object pooling for frequently created objects
```

```
class ObjectPool {
  constructor(createFn, resetFn) {
    this.createFn = createFn;
    this.resetFn = resetFn;
    this.pool = [];
  }

  get() {
    return this.pool.length > 0 ? this.pool.pop() : this.createFn();
  }

  release(obj) {
    this.resetFn(obj);
    this.pool.push(obj);
  }
}
```

Security Best Practices

```
// Avoid eval() and similar functions
// Bad
const userCode = getUserInput();
eval(userCode); // Never do this!

// Use strict mode to catch common mistakes
("use strict");

// Validate and sanitize user input
function sanitizeInput(input) {
  return input.replace(
    /<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi,
    ""
  );
}

// Use Content Security Policy headers
// Set in HTTP headers: Content-Security-Policy: default-src 'self'

// Avoid exposing sensitive data in client-side code
// Bad
const API_SECRET = "secret-key-123"; // Visible to everyone

// Good - keep secrets on server
const API_ENDPOINT = "/api/secure-endpoint"; // Server handles authentication

// Use HTTPS for all external requests
// Bad
fetch("http://api.example.com/data");

// Good
fetch("https://api.example.com/data");
```

Testing Best Practices

```javascript
// Write testable functions
// Bad - hard to test
function processOrder() {
  const order = getOrderFromDatabase();
  const user = getCurrentUser();
  const result = calculateTotal(order, user);
  updateUI(result);
  sendEmail(user.email, result);
}

// Good - testable
function calculateOrderTotal(order, user) {
  // Pure function - easy to test
  let total = order.items.reduce((sum, item) => sum + item.price, 0);
  if (user.isPremium) {
    total *= 0.9; // 10% discount
  }
  return total;
}

// Use descriptive test names
// Bad
test("user test", () => {
  /* ... */
});

// Good
test("should calculate 10% discount for premium users", () => {
  const order = { items: [{ price: 100 }] };
  const user = { isPremium: true };
  const total = calculateOrderTotal(order, user);
  expect(total).toBe(90);
});

// Mock external dependencies
// Using a mock for testing
const mockEmailService = {
  send: jest.fn().mockResolvedValue(true),
};

test("should send welcome email after user registration", async () => {
  await registerUser(userData, mockEmailService);
  expect(mockEmailService.send).toHaveBeenCalledWith(
    userData.email,
    "Welcome!"
  );
});
```

Documentation and Comments

```javascript
/**
 * Calculates the area of a circle
 * @param {number} radius - The radius of the circle
 * @returns {number} The area of the circle
 * @throws {Error} When radius is negative
 */
function calculateCircleArea(radius) {
  if (radius < 0) {
    throw new Error("Radius cannot be negative");
  }
  return Math.PI * radius * radius;
}

// Explain WHY, not WHAT
// Bad comment
let price = product.price * 0.9; // Multiply price by 0.9

// Good comment
let price = product.price * 0.9; // Apply 10% discount for early birds

// TODO comments for future improvements
// TODO: Implement caching for better performance
// FIXME: Handle edge case when user has no email
// HACK: Temporary workaround until API is fixed
```

Modern JavaScript (ES2020+)

Optional Chaining (?.)

```javascript
const user = {
  profile: {
    name: "John",
    address: {
      street: "123 Main St",
    },
  },
};

// Old way
const street =
  user && user.profile && user.profile.address && user.profile.address.street;

// With optional chaining
const street = user?.profile?.address?.street;
const phone = user?.profile?.contact?.phone ?? "No phone";

// With arrays
```

```
const firstFriend = user?.friends?.[0]?.name;

// With function calls
user?.profile?.getName?.();
```

Nullish Coalescing (??)

```
// Different from || operator
const value1 = 0 || "default"; // 'default' (0 is falsy)
const value2 = 0 ?? "default"; // 0 (only null/undefined trigger ??)

const value3 = "" || "default"; // 'default' (empty string is falsy)
const value4 = "" ?? "default"; // '' (only null/undefined trigger ??)

// Useful for default values
function createUser(options = {}) {
  return {
    name: options.name ?? "Anonymous",
    age: options.age ?? 0,
    active: options.active ?? true,
  };
}
```

BigInt

```
// For integers larger than Number.MAX_SAFE_INTEGER
const bigNumber = 1234567890123456789012345678901234567890n;
const anotherBig = BigInt("1234567890123456789012345678901234567890");

// Operations
const sum = 123n + 456n; // 579n
const product = 123n * 456n;

// Cannot mix BigInt with regular numbers
// const mixed = 123n + 456; // TypeError
const mixed = 123n + BigInt(456); // Correct way

// Converting
const regular = Number(123n); // 123 (loses precision for large numbers)
const bigFromRegular = BigInt(123); // 123n
```

Dynamic Imports

```
// Static import (must be at top level)
import { utils } from "./utils.js";
```

```
// Dynamic import (can be used anywhere)
async function loadUtilities() {
  if (someCondition) {
    const { utils } = await import("./utils.js");
    return utils;
  }
}

// Conditional loading
button.addEventListener("click", async () => {
  const { heavyLibrary } = await import("./heavy-library.js");
  heavyLibrary.doSomething();
});

// Error handling with dynamic imports
try {
  const module = await import("./maybe-missing-module.js");
  module.doSomething();
} catch (error) {
  console.error("Failed to load module:", error);
}
```

Private Class Fields

```
class BankAccount {
  // Private fields
  #balance = 0;
  #accountNumber;

  // Private method
  #validateAmount(amount) {
    return amount > 0 && typeof amount === "number";
  }

  constructor(accountNumber) {
    this.#accountNumber = accountNumber;
  }

  deposit(amount) {
    if (this.#validateAmount(amount)) {
      this.#balance += amount;
    } else {
      throw new Error("Invalid amount");
    }
  }

  getBalance() {
    return this.#balance;
  }

  // Static private field
  static #bankName = "My Bank";

  static getBankName() {
    return this.#bankName;
  }
}

const account = new BankAccount("123456");
account.deposit(1000);
console.log(account.getBalance()); // 1000
console.log(BankAccount.getBankName()); // 'My Bank'

// Attempting to access private fields directly will fail
// console.log(account.#balance); // SyntaxError
```

End of Cheatsheet

This cheatsheet covers the most important and modern features of JavaScript, from the basics to advanced concepts. For more details, always refer to the MDN Web Docs or the ECMAScript specification.

Happy coding!