```
JAVASCRIPT COMPLETE GUIDE - ZERO TO HERO 🚀
==========================================
// JS = Website interactive banata hai, server pe chalti hai, mobile apps banati hai
// Use cases: buttons, forms, games, APIs, React/Node.js, high salary skill

TABLE OF CONTENTS:
1.  Variables & Data Types (var, let, const)
2.  Operators & Expressions
3.  Control Structures (if/else, loops)
4.  Functions (basic to advanced)
5.  Objects & Arrays
6.  String Methods & Manipulation
7.  Array Methods & Iteration
8.  ES6+ Features
9.  DOM Manipulation
10.  Event Handling
11.  Asynchronous JavaScript (Promises, Async/Await)
12.  Error Handling
13.  Regular Expressions
14.  Classes & OOP
15.  Modules & Import/Export
16.  Advanced Concepts (Closures, Hoisting, etc.)
17.  Design Patterns
18.  Performance Optimization
19.  Modern JavaScript Frameworks Concepts
20.  Best Practices & Tips

==========================================================

1.  VARIABLES & DATA TYPES □
============================
var oldWay = "function-scoped";  // Avoid - purana hai
let modernWay = "block-scoped";  // Use this - modern
const PI = 3.14;  // Cannot change - constants ke liye

// Data Types - 7 main types
let num = 42;  // Number
let str = "Hello";  // String
let bool = true;  // Boolean
let undef;  // undefined
let empty = null;  // null
let sym = Symbol("id");  // Symbol
let big = 123n;  // BigInt

// Type checking
console.log(typeof num);  // "number"
console.log(typeof str);  // "string"

// Scope example
function test() {
    if (true) {
        var varScope = "function level";  // Accessible outside if
```

```
        let letScope = "block level";  // Not accessible outside if
    }
    console.log(varScope);  // Works
    // console.log(letScope);  // Error
}

// Hoisting - var moves to top
console.log(x);  // undefined (not error)
var x = 5;  // Actually: var x; console.log(x); x = 5;

=======================================================

2.  OPERATORS 
===============
// Arithmetic
let a = 10, b = 3;
console.log(a + b);  // 13 - addition
console.log(a - b);  // 7 - subtraction
console.log(a * b);  // 30 - multiplication
console.log(a / b);  // 3.33 - division
console.log(a % b);  // 1 - remainder
console.log(a ** b);  // 1000 - power

// Assignment shortcuts
let x = 5;
x += 3;  // x = x + 3 = 8
x -= 2;  // x = x - 2 = 6
x *= 2;  // x = x * 2 = 12
x++;  // x = x + 1
x--;  // x = x - 1

// Comparison
console.log(5 == "5");  // true - loose equality
console.log(5 === "5");  // false - strict equality
console.log(5 != "5");  // false
console.log(5 !== "5");  // true

// Logical
let p = true, q = false;
console.log(p && q);  // false - AND
console.log(p || q);  // true - OR
console.log(!p);  // false - NOT

// Ternary operator
let age = 18;
let status = age >= 18 ? "adult" : "minor";

// Nullish coalescing
let user = null;
let name = user ?? "Guest";  // If null/undefined use "Guest"

// Optional chaining
let obj = { user: { name: "John" } };
console.log(obj?.user?.name);  // "John" - safe access
```

```
console.log(obj?.user?.age);  // undefined - no error

=======================================================

3.  CONTROL FLOW 🔄
==================
// if-else
let score = 85;
if (score >= 90) console.log("A");
else if (score >= 80) console.log("B");
else console.log("C");

// switch
let day = "Monday";
switch (day) {
    case "Monday": console.log("Work"); break;
    case "Sunday": console.log("Rest"); break;
    default: console.log("Regular day");
}

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

// while loop
let count = 0;
while (count < 3) {
    console.log(count++);  // 0,1,2
}

// do-while (runs at least once)
let num = 0;
do {
    console.log(num++);
} while (num < 2);

// for-in (objects)
let person = { name: "John", age: 30 };
for (let key in person) {
    console.log(key, person[key]);  // name John, age 30
}

// for-of (arrays)
let arr = [1, 2, 3];
for (let val of arr) {
    console.log(val);  // 1,2,3
}

// break & continue
for (let i = 0; i < 10; i++) {
    if (i === 3) continue;  // skip 3
    if (i === 7) break;  // stop at 7
    console.log(i);  // 0,1,2,4,5,6
```

```
}

=======================================================

4.  FUNCTIONS 
==============
function greet(name) {
    return "Hello, " + name + "!";
}

// Function Expression
const greetExpression = function(name) {
    return "Hi, " + name + "!";
};

// Arrow Functions
const greetArrow = (name) => "Hey, " + name + "!";
const greetArrowMultiline = (name) => {
    return "Greetings, " + name + "!";
};

// Function with Default Parameters
function greetWithDefault(name = "World") {
    return "Hello, " + name + "!";
}

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

// Spread Operator
let numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // 3

// Higher-Order Functions
function applyOperation(x, y, operation) {
    return operation(x, y);
}
const add = (a, b) => a + b;
const multiply = (a, b) => a * b;

console.log(applyOperation(5, 3, add));      // 8
console.log(applyOperation(5, 3, multiply)); // 15

// Immediately Invoked Function Expression (IIFE)
(function() {
    console.log("IIFE executed!");
})();

// Recursive Function
function factorial(n) {
    if (n <= 1) return 1;
```

```
    return n * factorial(n - 1);
}
console.log(factorial(5)); // 120

// Callback Functions
function processData(data, callback) {
    let processed = data.map(x => x * 2);
    callback(processed);
}
processData([1, 2, 3], (result) => console.log(result));
```

==========

5. OBJECTS & ARRAYS

==========

```
// Object Creation
let person1 = {
    name: "John",
    age: 30,
    city: "New York",
    greet: function() {
        return "Hello, I'm " + this.name;
    }
};

// Object Constructor
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        return "Hi, I'm " + this.name;
    };
}
let person2 = new Person("Alice", 25);

// Object.create()
let personPrototype = {
    greet: function() {
        return "Hello from " + this.name;
    }
};
let person3 = Object.create(personPrototype);
person3.name = "Bob";

// Object Methods
let car = { brand: "Toyota", model: "Camry" };
console.log(Object.keys(car));        // ["brand", "model"]
console.log(Object.values(car));      // ["Toyota", "Camry"]
console.log(Object.entries(car));     // [["brand", "Toyota"], ["model",
"Camry"]]

// Object Destructuring
let { name, age } = person1;
```

```
console.log(name, age); // "John" 30

// Array Creation and Manipulation
let fruits = ["apple", "banana", "orange"];
let numbers = new Array(1, 2, 3, 4, 5);
let mixedArray = [1, "hello", true, null, { key: "value" }];

// Array Methods
fruits.push("grape");           // Add to end
fruits.pop();                   // Remove from end
fruits.unshift("mango");        // Add to beginning
fruits.shift();                 // Remove from beginning
fruits.splice(1, 1, "kiwi");    // Remove 1 element at index 1, add "kiwi"

// Array Destructuring
let [first, second, ...rest] = fruits;
console.log(first, second, rest);

// Nested Objects and Arrays
let company = {
    name: "Tech Corp",
    employees: [
        { name: "John", position: "Developer" },
        { name: "Jane", position: "Designer" }
    ],
    address: {
        street: "123 Tech St",
        city: "Silicon Valley"
    }
};

=======================================================

6.  FUNCTIONS □
===============
// Function declaration
function greet(name) {
    return "Hello " + name;  // return value
}

// Function expression
const greetExp = function(name) {
    return "Hi " + name;
};

// Arrow function (modern)
const greetArrow = (name) => "Hey " + name;
const greetMulti = (name) => {
    return "Greetings " + name;
};

// Default parameters
function greetDefault(name = "World") {
    return "Hello " + name;
```

```
}

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

// Spread operator
let nums = [1, 2, 3];
console.log(Math.max(...nums));  // 3

// Higher-order functions
function applyOp(x, y, operation) {
    return operation(x, y);
}
const add = (a, b) => a + b;
console.log(applyOp(5, 3, add));  // 8

// IIFE (Immediately Invoked Function Expression)
(function() {
    console.log("IIFE executed!");
})();

// Recursive function
function factorial(n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

// Callback function
function processData(data, callback) {
    let processed = data.map(x => x * 2);
    callback(processed);
}

============================================================

7.  OBJECTS & ARRAYS
======================
// Object creation
let person = {
    name: "John",
    age: 30,
    greet: function() { return "Hello " + this.name; }
};

// Object constructor
function Person(name, age) {
    this.name = name;
    this.age = age;
}
let person2 = new Person("Alice", 25);
```

```
// Object methods
let car = { brand: "Toyota", model: "Camry" };
console.log(Object.keys(car));  // ["brand", "model"]
console.log(Object.values(car));  // ["Toyota", "Camry"]
console.log(Object.entries(car));  // [["brand", "Toyota"], ["model",
"Camry"]]

// Object destructuring
let { name, age } = person;

// Array creation
let fruits = ["apple", "banana", "orange"];
let numbers = new Array(1, 2, 3, 4, 5);

// Array methods
fruits.push("grape");  // add to end
fruits.pop();  // remove from end
fruits.unshift("mango");  // add to beginning
fruits.shift();  // remove from beginning

// Array destructuring
let [first, second, ...rest] = fruits;
```

==========

8. STRING METHODS 📝

==========

```
let text = "JavaScript is Awesome";

// Basic methods
console.log(text.length);  // 21
console.log(text.charAt(0));  // "J"
console.log(text.indexOf("Script"));  // 4
console.log(text.slice(0, 10));  // "JavaScript"
console.log(text.toLowerCase());  // "javascript is awesome"
console.log(text.toUpperCase());  // "JAVASCRIPT IS AWESOME"

// Search and replace
console.log(text.includes("Script"));  // true
console.log(text.startsWith("Java"));  // true
console.log(text.replace("Awesome", "Amazing"));  // replace

// Split and join
let words = text.split(" ");  // ["JavaScript", "is", "Awesome"]
console.log(words.join("-"));  // "JavaScript-is-Awesome"

// Template literals
let name = "Alice", age = 25;
let message = `Hello, my name is ${name} and I'm ${age} years old.`;
```

==========

9. ARRAY METHODS 🔄

==========

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

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

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

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

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

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

// some - check if any matches
let hasEven = numbers.some(num => num % 2 === 0);  // true

// every - check if all match
let allPositive = numbers.every(num => num > 0);  // true

// Chaining methods
let result = numbers
    .filter(num => num % 2 === 0)
    .map(num => num * 3)
    .reduce((acc, num) => acc + num, 0);
```

==========

10. ES6+ FEATURES ⚡

==========

```
// Destructuring
let person = { name: "John", age: 30 };
let { name, age } = person;
let [a, b, c] = [1, 2, 3];

// Rest/Spread
let [head, ...tail] = [1, 2, 3, 4];  // head=1, tail=[2,3,4]
let arr1 = [1, 2], arr2 = [3, 4];
let combined = [...arr1, ...arr2];  // [1,2,3,4]

// Object shorthand
let name2 = "Alice", age2 = 25;
let person2 = { name2, age2 };

// Template literals
let greeting = `Hello ${name}!`;

// Arrow functions
const add = (a, b) => a + b;
const multiply = (a, b) => a * b;
```

```
// Let/const vs var
if (true) {
    let blockScoped = "only here";
    const constant = "can't change";
}

// Set and Map
let mySet = new Set([1, 2, 3, 3]);  // Set {1, 2, 3}
let myMap = new Map();
myMap.set("key", "value");
```

==========

11. DOM MANIPULATION 🌐

==========

```
// Selecting elements
let el = document.getElementById("myId");
let els = document.getElementsByClassName("myClass");
let query = document.querySelector(".myClass");
let queryAll = document.querySelectorAll(".myClass");

// Creating elements
let div = document.createElement("div");
div.textContent = "Hello";
div.innerHTML = "<strong>Bold</strong>";

// Modifying elements
el.textContent = "New text";
el.style.color = "red";
el.classList.add("newClass");
el.classList.remove("oldClass");
el.classList.toggle("active");

// Attributes
el.setAttribute("data-id", "123");
let value = el.getAttribute("data-id");

// Adding/removing elements
parent.appendChild(div);
parent.removeChild(el);
```

==========

12. EVENT HANDLING 🎯

==========

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

// Arrow function
button.addEventListener("click", () => console.log("Clicked!"));

// Event object
```

```
input.addEventListener("keydown", function(e) {
    console.log("Key:", e.key);
    console.log("Ctrl:", e.ctrlKey);
});

// Form events
form.addEventListener("submit", handleSubmit);
input.addEventListener("change", handleChange);

// Window events
window.addEventListener("load", () => console.log("Loaded"));
window.addEventListener("resize", () => console.log("Resized"));

// Event delegation
document.addEventListener("click", function(e) {
    if (e.target.classList.contains("btn")) {
        console.log("Button clicked");
    }
});

// Remove event listener
function handler() { console.log("Click"); }
button.addEventListener("click", handler);
button.removeEventListener("click", handler);
```

==========

13. ASYNC JAVASCRIPT □

==========

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

// Promises
let promise = new Promise((resolve, reject) => {
    let success = true;
    success ? resolve("Success") : reject("Error");
});

promise.then(result => console.log(result))
       .catch(error => console.log(error));

// Async/await
async function fetchAsync() {
    try {
        let response = await fetch("api/data");
        let data = await response.json();
        return data;
    } catch (error) {
        console.error(error);
    }
}
```

```
// Promise methods
Promise.all([promise1, promise2]).then(results => console.log(results));
Promise.race([promise1, promise2]).then(result => console.log(result));

// setTimeout/setInterval
setTimeout(() => console.log("After 2s"), 2000);
let interval = setInterval(() => console.log("Every 1s"), 1000);
clearInterval(interval);
```

==========

14. ERROR HANDLING ✕

==========

```
// Try-catch-finally
try {
    let result = riskyOperation();
    return result;
} catch (error) {
    console.error("Error:", error.message);
    return null;
} finally {
    console.log("Always runs");
}

// Throwing errors
function divide(a, b) {
    if (b === 0) throw new Error("Division by zero");
    return a / b;
}

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

// Async error handling
async function handleAsync() {
    try {
        let data = await fetchData();
        return data;
    } catch (error) {
        console.log("Async error:", error);
    }
}
```

==========

15. REGULAR EXPRESSIONS 🔍

==========

```
// Basic regex
let regex = /pattern/flags;
```

```
let regex2 = new RegExp("pattern", "flags");

// Common patterns
let email = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
let phone = /^\d{3}-\d{3}-\d{4}$/;

// String methods with regex
let text = "Hello 123 World";
console.log(text.match(/\d+/g));  // ["123"]
console.log(text.replace(/\d+/g, "XXX"));  // "Hello XXX World"

// Regex methods
let pattern = /\d+/;
console.log(pattern.test("123"));  // true
console.log(pattern.exec("abc123"));  // ["123"]

// Character classes
/\d/;  // digits
/\w/;  // word characters
/\s/;  // whitespace

// Quantifiers
/a+/;  // one or more
/a*/;  // zero or more
/a?/;  // zero or one
/a{3}/;  // exactly 3
```

==========

16. CLASSES & OOP

```
// Basic class
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        return `${this.name} makes sound`;
    }

    static getKingdom() {
        return "Animalia";
    }

    get info() {
        return `Animal: ${this.name}`;
    }
}

// Inheritance
class Dog extends Animal {
    constructor(name, breed) {
        super(name);
```

```
        this.breed = breed;
    }

    speak() {
        return `${this.name} barks`;
    }
}

// Usage
let dog = new Dog("Buddy", "Golden");
console.log(dog.speak());  // "Buddy barks"
console.log(Animal.getKingdom());  // "Animalia"

// Private fields (ES2022)
class BankAccount {
    #balance = 0;

    deposit(amount) {
        this.#balance += amount;
    }

    getBalance() {
        return this.#balance;
    }
}
```

==========

17. MODULES

```
// Named exports (utils.js)
export const PI = 3.14;
export function add(a, b) { return a + b; }
export class Calculator {}

// Default export (math.js)
export default function subtract(a, b) { return a - b; }

// Importing (main.js)
import subtract from './math.js';  // default
import { PI, add } from './utils.js';  // named
import * as Utils from './utils.js';  // namespace

// Dynamic imports
async function loadModule() {
    const module = await import('./utils.js');
    console.log(module.PI);
}

// Module pattern (old way)
const Module = (function() {
    let private = 0;
    return {
        public: function() { return private++; }
```

```
    };
})();

============================================================

18.  ADVANCED CONCEPTS 🚀
========================
// Closures
function outer(x) {
    return function inner(y) {
        return x + y;  // inner has access to x
    };
}
let addFive = outer(5);
console.log(addFive(3));  // 8

// Hoisting
console.log(hoisted());  // "Works"
function hoisted() { return "Works"; }

// This binding
let obj = {
    name: "Object",
    regular: function() { console.log(this.name); },  // "Object"
    arrow: () => { console.log(this.name); }  // undefined
};

// Call, apply, bind
function greet(greeting) {
    return greeting + " " + this.name;
}
let person = { name: "John" };
console.log(greet.call(person, "Hello"));  // "Hello John"
console.log(greet.apply(person, ["Hi"]));  // "Hi John"
let bound = greet.bind(person);

// Currying
function curry(fn) {
    return function curried(...args) {
        if (args.length >= fn.length) {
            return fn.apply(this, args);
        }
        return function(...args2) {
            return curried.apply(this, args.concat(args2));
        };
    };
}

// Debouncing
function debounce(func, wait) {
    let timeout;
    return function(...args) {
        clearTimeout(timeout);
        timeout = setTimeout(() => func.apply(this, args), wait);
```

```
    };
}

// Throttling
function throttle(func, limit) {
    let inThrottle;
    return function(...args) {
        if (!inThrottle) {
            func.apply(this, args);
            inThrottle = true;
            setTimeout(() => inThrottle = false, limit);
        }
    };
}
```

==========

19. DESIGN PATTERNS 🎨

==========

```
// Singleton
class Singleton {
    constructor() {
        if (Singleton.instance) return Singleton.instance;
        Singleton.instance = this;
    }
}

// Factory
class CarFactory {
    createCar(type) {
        switch(type) {
            case 'sedan': return new Sedan();
            case 'suv': return new SUV();
        }
    }
}

// Observer
class Subject {
    constructor() { this.observers = []; }
    subscribe(observer) { this.observers.push(observer); }
    notify(data) { this.observers.forEach(obs => obs.update(data)); }
}

// Module
const Calculator = (function() {
    let result = 0;
    return {
        add: function(x) { result += x; return this; },
        getResult: function() { return result; }
    };
})();
```

==========

```
20.  PERFORMANCE 
==================
// Efficient DOM updates
const fragment = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
    const div = document.createElement('div');
    fragment.appendChild(div);
}
document.body.appendChild(fragment);

// Lazy loading
function lazy(fn) {
    let loaded = false, result;
    return function() {
        if (!loaded) {
            result = fn.apply(this, arguments);
            loaded = true;
        }
        return result;
    };
}

// Memory management
function cleanup() {
    element.removeEventListener('click', handler);  // remove listeners
    clearTimeout(timeoutId);  // clear timeouts
    largeObject = null;  // release references
}

// Use efficient array methods
const result = array
    .filter(item => item.active)  // filter first
    .map(item => item.name)  // then transform
    .slice(0, 10);  // limit results

============================================================

21.  MODERN FRAMEWORKS CONCEPTS 
==================================
// Component pattern
class Component {
    constructor(props) {
        this.props = props;
        this.state = {};
    }
    setState(newState) {
        this.state = { ...this.state, ...newState };
        this.render();
    }
}

// State management
class Store {
```

```
    constructor(reducer, initialState) {
        this.reducer = reducer;
        this.state = initialState;
        this.listeners = [];
    }
    dispatch(action) {
        this.state = this.reducer(this.state, action);
        this.listeners.forEach(listener => listener());
    }
}

// Observable pattern
class Observable {
    constructor(fn) { this.fn = fn; }
    subscribe(observer) { return this.fn(observer); }
    map(mapFn) {
        return new Observable(observer => {
            return this.subscribe({
                next: value => observer.next(mapFn(value))
            });
        });
    }
}
```

==========

22. BEST PRACTICES ✅

==========

```
"use strict";  // always use strict mode

// Naming conventions
const CONSTANTS = "UPPER_CASE";
const camelCase = "preferred";
const PascalCase = class {};

// Use const by default
const immutable = "won't change";
let mutable = "might change";

// Meaningful names
const currentDate = new Date();  // good
const d = new Date();  // bad

// Single responsibility functions
function calculateTax(price, rate) { return price * rate; }
function formatCurrency(amount) { return `$${amount.toFixed(2)}`; }

// Pure functions (no side effects)
function add(a, b) { return a + b; }

// Use modern features
const street = user?.address?.street;  // optional chaining
const name = user.name ?? 'Anonymous';  // nullish coalescing
const { timeout = 5000 } = options;  // destructuring with defaults
```

```
// Error handling
async function safeOperation(data) {
    if (!data) throw new Error('Data required');
    try {
        return processData(data);
    } catch (error) {
        console.error('Operation failed:', error);
        throw error;
    }
}

// Performance tips
const memoized = memoize(expensiveFunction);  // cache results
const doubled = numbers.map(n => n * 2);  // use array methods
const { name, email } = user;  // destructuring

========================================================

QUICK REFERENCE CHEAT SHEET 📋
==============================

VARIABLES:
let name = "value";  // block-scoped
const PI = 3.14;  // constant
var old = "avoid";  // function-scoped

FUNCTIONS:
function name() {}  // declaration
const name = () => {};  // arrow
const name = function() {};  // expression

ARRAYS:
.push() .pop() .shift() .unshift()  // add/remove
.map() .filter() .reduce() .find()  // iteration
.forEach() .some() .every()  // utilities

OBJECTS:
Object.keys() .values() .entries()  // get properties
{ ...obj }  // spread operator
{ name, age }  // shorthand

PROMISES:
async/await  // modern async
.then() .catch()  // promise chains
Promise.all() Promise.race()  // utilities

DOM:
document.querySelector()  // select
element.addEventListener()  // events
element.classList.add()  // styling

REMEMBER:
-  Use === instead of ==
```

- Use const/let instead of var
- Handle errors with try/catch
- Use arrow functions for callbacks
- Destructure objects/arrays
- Use template literals `${}`
- Chain array methods efficiently
- Always validate user input
- Use meaningful variable names
- Keep functions small and focused

Happy Coding! 🎉🚀

```
// reverse - Reverse array
let reversed = [...numbers].reverse();

// includes - Check if array includes element
console.log(numbers.includes(5)); // true

// Array.from() - Create array from iterable
let arrayFromString = Array.from("hello"); // ["h", "e", "l", "l", "o"]

// Chaining Methods
let result = numbers
    .filter(num => num % 2 === 0)
    .map(num => num * 3)
    .reduce((acc, num) => acc + num, 0);
```

==========

8. ES6+ FEATURES
==========

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

let colors = ["red", "green", "blue"];
let [primary, secondary, tertiary] = colors;

// Default Values in Destructuring
let { x = 10, y = 20 } = { x: 5 };
console.log(x, y); // 5, 20

// Rest and Spread Operators
let [head, ...tail] = [1, 2, 3, 4, 5];
console.log(head); // 1
console.log(tail); // [2, 3, 4, 5]

let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

// Object Shorthand
let name2 = "Alice";
```

```
let age2 = 25;
let person4 = { name2, age2 }; // Same as { name2: name2, age2: age2 }

// Computed Property Names
let prop = "dynamicKey";
let obj2 = {
    [prop]: "dynamicValue",
    [`${prop}2`]: "anotherValue"
};

// Symbol
let sym1 = Symbol("description");
let sym2 = Symbol.for("globalSymbol");

// Set and Map
let mySet = new Set([1, 2, 3, 3, 4]);
console.log(mySet); // Set(4) {1, 2, 3, 4}

let myMap = new Map();
myMap.set("key1", "value1");
myMap.set("key2", "value2");

// WeakSet and WeakMap
let weakSet = new WeakSet();
let weakMap = new WeakMap();

// Generators
function* numberGenerator() {
    yield 1;
    yield 2;
    yield 3;
}
let gen = numberGenerator();
console.log(gen.next().value); // 1

// Proxy
let target = { name: "John" };
let proxy = new Proxy(target, {
    get(target, property) {
        console.log(`Accessing ${property}`);
        return target[property];
    }
});

==========================================================

9. DOM MANIPULATION
===================

// Selecting Elements
let elementById = document.getElementById("myId");
let elementsByClass = document.getElementsByClassName("myClass");
let elementsByTag = document.getElementsByTagName("div");
let querySelector = document.querySelector(".myClass");
```

```
let querySelectorAll = document.querySelectorAll(".myClass");

// Creating Elements
let newDiv = document.createElement("div");
newDiv.textContent = "Hello World";
newDiv.innerHTML = "<strong>Bold Text</strong>";

// Modifying Elements
elementById.textContent = "New Text";
elementById.innerHTML = "<em>Italic Text</em>";
elementById.style.color = "red";
elementById.style.backgroundColor = "yellow";

// Adding/Removing Classes
elementById.classList.add("newClass");
elementById.classList.remove("oldClass");
elementById.classList.toggle("activeClass");
elementById.classList.contains("someClass");

// Attributes
elementById.setAttribute("data-value", "123");
let attributeValue = elementById.getAttribute("data-value");
elementById.removeAttribute("data-value");

// Parent-Child Relationships
let parent = elementById.parentNode;
let children = elementById.children;
let firstChild = elementById.firstElementChild;
let lastChild = elementById.lastElementChild;

// Adding/Removing Elements
parent.appendChild(newDiv);
parent.insertBefore(newDiv, firstChild);
parent.removeChild(elementById);

// Clone Elements
let clonedElement = elementById.cloneNode(true);

// Form Handling
let form = document.getElementById("myForm");
let inputValue = document.getElementById("myInput").value;
form.addEventListener("submit", function(e) {
    e.preventDefault();
    // Handle form submission
});

=========================================================

10. EVENT HANDLING
==================

// Basic Event Listeners
button.addEventListener("click", function() {
    console.log("Button clicked!");
```

```
});

// Arrow Function Event Handler
button.addEventListener("click", () => {
    console.log("Arrow function handler");
});

// Event Object
input.addEventListener("keydown", function(event) {
    console.log("Key pressed:", event.key);
    console.log("Key code:", event.keyCode);
    console.log("Ctrl pressed:", event.ctrlKey);
});

// Mouse Events
element.addEventListener("mouseenter", handleMouseEnter);
element.addEventListener("mouseleave", handleMouseLeave);
element.addEventListener("mouseover", handleMouseOver);
element.addEventListener("mouseout", handleMouseOut);

// Form Events
form.addEventListener("submit", handleSubmit);
input.addEventListener("change", handleChange);
input.addEventListener("input", handleInput);
input.addEventListener("focus", handleFocus);
input.addEventListener("blur", handleBlur);

// Window Events
window.addEventListener("load", function() {
    console.log("Page fully loaded");
});

window.addEventListener("resize", function() {
    console.log("Window resized");
});

// Event Delegation
document.addEventListener("click", function(e) {
    if (e.target.classList.contains("button")) {
        console.log("Button clicked via delegation");
    }
});

// Removing Event Listeners
function clickHandler() {
    console.log("Clicked");
}
button.addEventListener("click", clickHandler);
button.removeEventListener("click", clickHandler);

// Custom Events
let customEvent = new CustomEvent("myEvent", {
    detail: { message: "Hello from custom event" }
});
```

```
element.dispatchEvent(customEvent);

element.addEventListener("myEvent", function(e) {
    console.log(e.detail.message);
});

============================================================

11. ASYNCHRONOUS JAVASCRIPT
============================

// Callbacks
function fetchData(callback) {
    setTimeout(() => {
        callback("Data fetched");
    }, 1000);
}

fetchData((data) => {
    console.log(data);
});

// Promises
let promise = new Promise((resolve, reject) => {
    let success = true;
    if (success) {
        resolve("Operation successful");
    } else {
        reject("Operation failed");
    }
});

promise
    .then(result => console.log(result))
    .catch(error => console.log(error));

// Promise Chaining
fetch("https://api.example.com/data")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error(error));

// Async/Await
async function fetchDataAsync() {
    try {
        let response = await fetch("https://api.example.com/data");
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

// Promise.all() - Wait for all promises
```

```
let promise1 = Promise.resolve(3);
let promise2 = new Promise(resolve => setTimeout(() => resolve("foo"),
1000));
let promise3 = Promise.resolve(42);

Promise.all([promise1, promise2, promise3])
    .then(values => console.log(values)); // [3, "foo", 42]

// Promise.race() - First to complete
Promise.race([promise1, promise2, promise3])
    .then(value => console.log(value)); // 3

// Promise.allSettled() - Wait for all, regardless of outcome
Promise.allSettled([promise1, promise2, promise3])
    .then(results => console.log(results));

// setTimeout and setInterval
setTimeout(() => {
    console.log("Executed after 2 seconds");
}, 2000);

let intervalId = setInterval(() => {
    console.log("Executed every second");
}, 1000);

// Clear interval
clearInterval(intervalId);

=========================================================

12. ERROR HANDLING
==================

// Try-Catch-Finally
function riskyOperation() {
    try {
        // Code that might throw an error
        let result = someFunction();
        return result;
    } catch (error) {
        console.error("Error occurred:", error.message);
        return null;
    } finally {
        console.log("This always executes");
    }
}

// Throwing Custom Errors
function divide(a, b) {
    if (b === 0) {
        throw new Error("Division by zero is not allowed");
    }
    return a / b;
}
```

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

function validateAge(age) {
    if (age < 0 || age > 150) {
        throw new ValidationError("Age must be between 0 and 150");
    }
}

// Error Handling with Async/Await
async function handleAsyncErrors() {
    try {
        let data = await fetchData();
        return data;
    } catch (error) {
        if (error instanceof TypeError) {
            console.log("Type error occurred");
        } else if (error instanceof ReferenceError) {
            console.log("Reference error occurred");
        } else {
            console.log("Unknown error:", error);
        }
    }
}

// Global Error Handling
window.addEventListener("error", function(e) {
    console.log("Global error:", e.error);
});

window.addEventListener("unhandledrejection", function(e) {
    console.log("Unhandled promise rejection:", e.reason);
});

============================================================

13. REGULAR EXPRESSIONS
=======================

// Basic Regex Creation
let regex1 = /pattern/flags;
let regex2 = new RegExp("pattern", "flags");

// Common Patterns
let emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
let phonePattern = /^\d{3}-\d{3}-\d{4}$/;
let urlPattern = /^https?:\/\/.+/;
```

```
// String Methods with Regex
let text = "The quick brown fox jumps over the lazy dog";
console.log(text.match(/o/g));        // ["o", "o", "o", "o"]
console.log(text.search(/fox/));      // 16
console.log(text.replace(/o/g, "0")); // Replace all 'o' with '0'

// Regex Methods
let pattern = /\d+/g;
console.log(pattern.test("123"));     // true
console.log(pattern.exec("abc123def")); // ["123"]

// Character Classes
let digitPattern = /\d/;      // Digits
let wordPattern = /\w/;       // Word characters
let spacePattern = /\s/;      // Whitespace
let notDigitPattern = /\D/;   // Non-digits

// Quantifiers
let oneOrMore = /a+/;         // One or more 'a'
let zeroOrMore = /a*/;        // Zero or more 'a'
let zeroOrOne = /a?/;         // Zero or one 'a'
let exactlyThree = /a{3}/;    // Exactly 3 'a's
let threeToFive = /a{3,5}/;   // 3 to 5 'a's

// Groups and Capturing
let namePattern = /(\w+)\s+(\w+)/;
let match = "John Doe".match(namePattern);
console.log(match[1]); // "John"
console.log(match[2]); // "Doe"

// Lookahead and Lookbehind
let positiveLookahead = /\d+(?=px)/;  // Digit followed by 'px'
let negativeLookahead = /\d+(?!px)/;  // Digit not followed by 'px'

============================================================

14. CLASSES & OOP
=================

// Basic Class Definition
class Animal {
    constructor(name, species) {
        this.name = name;
        this.species = species;
    }

    speak() {
        return `${this.name} makes a sound`;
    }

    // Static method
    static getKingdom() {
        return "Animalia";
    }
```

```
    // Getter
    get info() {
        return `${this.name} is a ${this.species}`;
    }

    // Setter
    set name(newName) {
        this._name = newName;
    }
}

// Inheritance
class Dog extends Animal {
    constructor(name, breed) {
        super(name, "Canine");
        this.breed = breed;
    }

    speak() {
        return `${this.name} barks`;
    }

    wagTail() {
        return `${this.name} wags tail`;
    }
}

// Creating Instances
let animal = new Animal("Generic", "Unknown");
let dog = new Dog("Buddy", "Golden Retriever");

console.log(dog.speak());    // "Buddy barks"
console.log(dog.info);       // Getter
console.log(Animal.getKingdom()); // Static method

// Private Fields (ES2022)
class BankAccount {
    #balance = 0;

    constructor(initialBalance) {
        this.#balance = initialBalance;
    }

    deposit(amount) {
        this.#balance += amount;
    }

    getBalance() {
        return this.#balance;
    }
}

// Mixin Pattern
```

```
let Flyable = {
    fly() {
        return `${this.name} is flying`;
    }
};

class Bird extends Animal {
    constructor(name) {
        super(name, "Avian");
    }
}

Object.assign(Bird.prototype, Flyable);

// Abstract Class Pattern
class Shape {
    constructor() {
        if (this.constructor === Shape) {
            throw new Error("Cannot instantiate abstract class");
        }
    }

    area() {
        throw new Error("Must implement area method");
    }
}

class Circle extends Shape {
    constructor(radius) {
        super();
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius * this.radius;
    }
}
```

==========

```
15. MODULES & IMPORT/EXPORT
============================

// Named Exports (utils.js)
export const PI = 3.14159;
export function add(a, b) {
    return a + b;
}
export class Calculator {
    multiply(a, b) {
        return a * b;
    }
}
```

```
// Default Export (math.js)
export default function subtract(a, b) {
    return a - b;
}

// Mixed Exports
export { PI as pi };
export { add as addition };

// Importing (main.js)
import subtract from './math.js';              // Default import
import { PI, add } from './utils.js';          // Named imports
import { PI as pi } from './utils.js';         // Aliased import
import * as Utils from './utils.js';           // Namespace import

// Dynamic Imports
async function loadModule() {
    const module = await import('./utils.js');
    console.log(module.PI);
}

// Re-exports
export { PI } from './utils.js';
export * from './utils.js';

// Module Pattern (Before ES6)
const ModulePattern = (function() {
    let privateVar = 0;

    function privateFunction() {
        return "Private";
    }

    return {
        publicMethod: function() {
            return privateFunction();
        },
        increment: function() {
            privateVar++;
        }
    };
})();

// CommonJS (Node.js)
// module.exports = { PI, add };
// const { PI, add } = require('./utils');

=======================================================

16. ADVANCED CONCEPTS
=====================

// Closures
function outerFunction(x) {
```

```
    return function innerFunction(y) {
        return x + y;
    };
}
let addFive = outerFunction(5);
console.log(addFive(3)); // 8

// Hoisting Examples
console.log(hoistedFunction()); // "Works!"
function hoistedFunction() {
    return "Works!";
}

// Event Loop and Call Stack
console.log("1");
setTimeout(() => console.log("2"), 0);
console.log("3");
// Output: 1, 3, 2

// Prototypal Inheritance
function Person(name) {
    this.name = name;
}
Person.prototype.greet = function() {
    return "Hello, " + this.name;
};

let person = new Person("Alice");
console.log(person.greet());

// This Binding
let obj = {
    name: "Object",
    regularFunction: function() {
        console.log(this.name); // "Object"
    },
    arrowFunction: () => {
        console.log(this.name); // undefined (lexical this)
    }
};

// Call, Apply, Bind
function greet(greeting, punctuation) {
    return greeting + " " + this.name + punctuation;
}
let person = { name: "John" };

console.log(greet.call(person, "Hello", "!"));
console.log(greet.apply(person, ["Hi", "."]));
let boundGreet = greet.bind(person);

// Currying
function curry(fn) {
    return function curried(...args) {
```

```
        if (args.length >= fn.length) {
            return fn.apply(this, args);
        } else {
            return function(...args2) {
                return curried.apply(this, args.concat(args2));
            };
        }
    };
}

const add = (a, b, c) => a + b + c;
const curriedAdd = curry(add);
console.log(curriedAdd(1)(2)(3)); // 6

// Memoization
function memoize(fn) {
    const cache = {};
    return function(...args) {
        const key = JSON.stringify(args);
        if (cache[key]) {
            return cache[key];
        }
        const result = fn.apply(this, args);
        cache[key] = result;
        return result;
    };
}

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

// Throttling
function throttle(func, limit) {
    let inThrottle;
    return function() {
        const args = arguments;
        const context = this;
        if (!inThrottle) {
            func.apply(context, args);
            inThrottle = true;
            setTimeout(() => inThrottle = false, limit);
        }
    };
}
```

```
=======================================================

17. DESIGN PATTERNS
====================

// Singleton Pattern
class Singleton {
    constructor() {
        if (Singleton.instance) {
            return Singleton.instance;
        }
        Singleton.instance = this;
    }
}

// Factory Pattern
class CarFactory {
    createCar(type) {
        switch(type) {
            case 'sedan':
                return new Sedan();
            case 'suv':
                return new SUV();
            default:
                throw new Error('Unknown car type');
        }
    }
}

// Observer Pattern
class Subject {
    constructor() {
        this.observers = [];
    }

    subscribe(observer) {
        this.observers.push(observer);
    }

    unsubscribe(observer) {
        this.observers = this.observers.filter(obs => obs !== observer);
    }

    notify(data) {
        this.observers.forEach(observer => observer.update(data));
    }
}

class Observer {
    update(data) {
        console.log("Observer received:", data);
    }
}
```

```
// Module Pattern
const Calculator = (function() {
    let result = 0;

    return {
        add: function(x) {
            result += x;
            return this;
        },
        multiply: function(x) {
            result *= x;
            return this;
        },
        getResult: function() {
            return result;
        }
    };
})();

// Decorator Pattern
function readonly(target, name, descriptor) {
    descriptor.writable = false;
    return descriptor;
}

class Example {
    @readonly
    method() {
        return "This method is readonly";
    }
}

// Strategy Pattern
class PaymentStrategy {
    pay(amount) {
        throw new Error("Must implement pay method");
    }
}

class CreditCardStrategy extends PaymentStrategy {
    pay(amount) {
        return `Paid ${amount} using Credit Card`;
    }
}

class PayPalStrategy extends PaymentStrategy {
    pay(amount) {
        return `Paid ${amount} using PayPal`;
    }
}

========================================================
```

18. PERFORMANCE OPTIMIZATION

```
// Efficient DOM Manipulation
function efficientDOMUpdate() {
    const fragment = document.createDocumentFragment();
    for (let i = 0; i < 1000; i++) {
        const div = document.createElement('div');
        div.textContent = `Item ${i}`;
        fragment.appendChild(div);
    }
    document.body.appendChild(fragment);
}

// Lazy Loading
function lazyLoad(fn) {
    let loaded = false;
    let result;

    return function() {
        if (!loaded) {
            result = fn.apply(this, arguments);
            loaded = true;
        }
        return result;
    };
}

// Web Workers (main thread)
const worker = new Worker('worker.js');
worker.postMessage({data: 'heavy computation'});
worker.onmessage = function(e) {
    console.log('Result from worker:', e.data);
};

// Event Delegation for Performance
document.addEventListener('click', function(e) {
    if (e.target.classList.contains('button')) {
        handleButtonClick(e.target);
    }
});

// Efficient Array Operations
// Use map instead of forEach when transforming
const transformed = array.map(item => item * 2);

// Use filter for conditional operations
const filtered = array.filter(item => item > 10);

// Use reduce for aggregations
const sum = array.reduce((acc, item) => acc + item, 0);

// Memory Management
function cleanupExample() {
```

```
    // Remove event listeners
    element.removeEventListener('click', handler);

    // Clear timeouts/intervals
    clearTimeout(timeoutId);
    clearInterval(intervalId);

    // Set references to null
    largeObject = null;
}

// Avoid Memory Leaks
// Bad: Creates closure that holds reference
function createHandler() {
    const largeData = new Array(1000000);
    return function() {
        // Uses largeData
    };
}

// Good: Release reference
function createHandlerOptimized() {
    const largeData = new Array(1000000);
    const result = processData(largeData);
    return function() {
        return result;
    };
}

============================================================

19. MODERN FRAMEWORK CONCEPTS
==============================

// Component-Based Architecture
class Component {
    constructor(props) {
        this.props = props;
        this.state = {};
    }

    setState(newState) {
        this.state = { ...this.state, ...newState };
        this.render();
    }

    render() {
        // Update DOM
    }
}

// Virtual DOM Concept
function createElement(type, props, ...children) {
    return {
```

```
        type,
        props: {
            ...props,
            children: children.map(child =>
                typeof child === "object" ? child :
createTextElement(child)
            )
        }
    };
}

// State Management Pattern
class Store {
    constructor(reducer, initialState) {
        this.reducer = reducer;
        this.state = initialState;
        this.listeners = [];
    }

    getState() {
        return this.state;
    }

    dispatch(action) {
        this.state = this.reducer(this.state, action);
        this.listeners.forEach(listener => listener());
    }

    subscribe(listener) {
        this.listeners.push(listener);
        return () => {
            this.listeners = this.listeners.filter(l => l !== listener);
        };
    }
}

// Reactive Programming
class Observable {
    constructor(fn) {
        this.fn = fn;
    }

    subscribe(observer) {
        return this.fn(observer);
    }

    map(mapFn) {
        return new Observable(observer => {
            return this.subscribe({
                next: value => observer.next(mapFn(value)),
                error: err => observer.error(err),
                complete: () => observer.complete()
            });
        });
```

```
    }
}

// Dependency Injection
class Container {
    constructor() {
        this.services = {};
    }

    register(name, definition) {
        this.services[name] = definition;
    }

    get(name) {
        const serviceDefinition = this.services[name];
        if (typeof serviceDefinition === 'function') {
            return serviceDefinition();
        }
        return serviceDefinition;
    }
}
```

==========

20. BEST PRACTICES & TIPS

==========

```
// Use Strict Mode
"use strict";

// Consistent Naming Conventions
const CONSTANTS_UPPER_CASE = "value";
const camelCaseVariables = "preferred";
const PascalCaseClasses = class {};

// Code Organization
// Group related functionality
const UserService = {
    create: (userData) => { /* ... */ },
    update: (id, userData) => { /* ... */ },
    delete: (id) => { /* ... */ },
    findById: (id) => { /* ... */ }
};

// Error Handling Best Practices
function safeOperation(data) {
    if (!data) {
        throw new Error('Data is required');
    }

    try {
        return processData(data);
    } catch (error) {
        console.error('Operation failed:', error);
```

```
        throw error; // Re-throw if needed
    }
}

// Use const by default, let when reassignment needed
const immutableValue = "won't change";
let mutableValue = "might change";

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

// 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);

// Function best practices
// Single responsibility
function calculateTax(price, rate) {
    return price * rate;
}

function formatCurrency(amount) {
    return `$${amount.toFixed(2)}`;
}

// Pure functions (no side effects)
function add(a, b) {
    return a + b; // No side effects
}

// Comments for complex logic
function complexAlgorithm(data) {
    // Step 1: Sort data by priority
    const sorted = data.sort((a, b) => b.priority - a.priority);

    // Step 2: Group by category
    const grouped = sorted.reduce((acc, item) => {
        if (!acc[item.category]) {
            acc[item.category] = [];
        }
        acc[item.category].push(item);
        return acc;
    }, {});

    return grouped;
}
```

```
// Performance considerations
// Use array methods efficiently
const result = data
    .filter(item => item.isValid)
    .map(item => transform(item))
    .slice(0, 10); // Limit results

// Avoid unnecessary operations
// Cache expensive calculations
const memoizedExpensiveFunction = memoize(expensiveFunction);

// Use object/array destructuring
const { name, email } = user;
const [first, second] = array;

// Template literals for string concatenation
const message = `Hello ${name}, your email is ${email}`;

// Use array methods instead of loops when appropriate
const doubled = numbers.map(n => n * 2);
const evens = numbers.filter(n => n % 2 === 0);
const sum = numbers.reduce((acc, n) => acc + n, 0);

// Async/await for better readability
async function fetchUserData(id) {
    try {
        const user = await api.getUser(id);
        const profile = await api.getProfile(user.profileId);
        return { user, profile };
    } catch (error) {
        console.error('Failed to fetch user data:', error);
        throw error;
    }
}

// Use modern JavaScript features
// Optional chaining
const street = user?.address?.street;

// Nullish coalescing
const name = user.name ?? 'Anonymous';

// Destructuring with defaults
const { timeout = 5000 } = options;

============================================================

SUMMARY OF KEY CONCEPTS:
========================

1.  Variables: var (function-scoped), let/const (block-scoped)
2.  Functions: Declarations, expressions, arrow functions, higher-order
3.  Objects: Literals, constructors, prototypes, destructuring
4.  Arrays: Methods, iteration, transformation, filtering
```

5. Asynchronous: Callbacks, promises, async/await
6. Classes: ES6 classes, inheritance, static methods
7. Modules: Import/export, dynamic imports
8. Advanced: Closures, hoisting, this binding, prototypes
9. DOM: Selection, manipulation, events
10. Error Handling: Try/catch, custom errors
11. Regex: Patterns, methods, validation
12. Performance: Optimization techniques, memory management
13. Modern Features: ES6+, destructuring, spread/rest
14. Design Patterns: Singleton, factory, observer, module
15. Best Practices: Clean code, naming, organization

REMEMBER:
- Practice regularly with coding challenges
- Build projects to apply concepts
- Read documentation and stay updated
- Use debugging tools and console effectively
- Write clean, readable, and maintainable code
- Test your code thoroughly
- Consider performance implications
- Follow coding standards and conventions

Happy Learning JavaScript! 🚀