Lectures on JavaScript: Fundamentals and Beyond

November 4, 2025

Contents

1 Introduction to JavaScript

JavaScript (JS) is a high-level, interpreted programming language primarily used for web development. It enables interactive web pages by manipulating the Document Object Model (DOM). Created by Brendan Eich in 1995, it has evolved into a versatile language powering both client-side and server-side applications (via Node.js).

1.1 What is JavaScript?

JavaScript is a dynamic, prototype-based language that supports both functional and imperative programming paradigms. It is interpreted, meaning code is executed line-by-line without prior compilation, allowing for quick prototyping but potential runtime errors.

Key characteristics:

- **Interpreted**: Executed directly by the engine (e.g., V8 in Chrome).
- **Prototype-based**: Objects inherit properties and methods from prototypes, enabling flexible object-oriented programming without classes (though classes were added in ES6).
- **Functional and Imperative**: Supports pure functions, closures, and higher-order functions alongside traditional loops and conditionals.
- **Dynamic Typing**: Variables are declared without specifying types; types are inferred at runtime, promoting flexibility but requiring careful type checking.

1.2 Setting Up Your Environment

To begin coding in JavaScript:

1. Open your browser's developer console (F12 in most browsers) for immediate execution.
2. Install Node.js from nodejs.org for running JS outside the browser: Use `npm install -g node` in your terminal.
3. Utilize online platforms like JSFiddle, CodePen, or Replit for quick testing without local setup.

2 Variables and Data Types

Variables are containers for storing data values. In JavaScript, declare them using `var`, `let`, or `const`, introduced to improve scoping and prevent common bugs.

2.1 Declaration and Scope

Scope determines variable accessibility. JavaScript has evolved from function scope to block scope for better modularity.

`var` Function-scoped and hoisted (moved to the top of the function during compilation), which can lead to unexpected behavior like using variables before declaration.

`let` Block-scoped (limited to {}), not hoisted, ideal for counters in loops.

`const` Block-scoped, cannot be reassigned after initialization (though mutable objects like arrays can have their contents changed).

```
var globalVar = 'I am global';  // Accessible everywhere in the function
function example() {
    if (true) {
        var functionScoped = 'Var inside if, but accessible outside';
        let blockScoped = 'Let inside if, not accessible outside';
        const constant = 'Cannot reassign';
    }
    console.log(functionScoped);  // Works
    // console.log(blockScoped);  // Error: not defined
}
```

Listing 1: Variable Declarations and Scope Example

Explanation: In the example, `var` leaks out of the block, while `let` and `const` respect the if-block's boundaries, preventing scope pollution.

2.2 Primitive Data Types

JavaScript features seven primitive (immutable) data types, which are the building blocks for all values. Primitives are passed by value, unlike objects which are passed by reference.

Undefined Represents a variable that has been declared but not assigned a value; default return for non-existent object properties.

Null Explicitly indicates no value or empty object; often used for intentional absence.

Boolean Represents logical values: `true` or `false`; coerced from other types in conditionals (e.g., non-empty string is true).

String Immutable sequences of characters for text representation; supports single/double quotes or template literals (ES6).

Number Represents integers and floating-point numbers; uses IEEE 754 double precision, with `NaN` for invalid operations.

BigInt Handles arbitrarily large integers beyond safe integer limits ($2^{53} - 1$), appended with 'n' (e.g., 1n).

Symbol Unique, immutable primitives used as object keys to avoid property name collisions; introduced in ES6.

Non-primitives include Objects (encompassing arrays, functions, and more), which are mutable collections of key-value pairs.

```
let undef;                    // undefined
let nul = null;              // null
let bool = true;             // boolean
let str = 'Hello, World!';   // string (explained below)
let num = 42.5;              // number
let big = 1234567890123456789012345678901234567890n;  // BigInt
let sym = Symbol('unique');  // Symbol
```

Listing 2: Primitive Data Types Example

Explanation: This code initializes each primitive. Note how `undef` automatically gets `undefined`. Use `typeof` operator to check types: `typeof str` returns 'string'.

2.2.1 Strings

Strings are sequences of Unicode characters used for text data. They are immutable, meaning once created, their content cannot be changed directly (though new strings can be derived). Strings support various methods for manipulation like concatenation, slicing, and searching.

Key features:

- Escaped characters: `\n` for newline, `\'` for single quote.
- Length property: `str.length` gives character count.
- Methods: `toUpperCase()`, `includes('substring')`, `slice(start, end)`.

```
let greeting = 'Hello';
let name = 'Brenda';
let full = greeting + ', ' + name + '!';  // Concatenation: 'Hello, Brenda!'
let upper = full.toUpperCase();           // 'HELLO, Brenda!'
let sub = full.slice(7, 11);              // 'Brenda'
console.log(full.includes('Brenda'));        // true
```

Listing 3: String Manipulation Example

Explanation: Strings are concatenated with `+` (coerces non-strings). Methods return new strings without altering the original, preserving immutability. For modern interpolation, use template literals (see ES6 section).

3 Operators and Control Structures

Operators perform operations on values, while control structures dictate code flow based on conditions or repetitions.

3.1 Arithmetic and Comparison Operators

Arithmetic operators handle math; comparison checks equality/inequality. JavaScript uses loose (`==`) and strict (`===`) equality to handle type coercion.

Arithmetic `+` (addition/concat), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo/remainder), `**` (exponentiation, ES2016).

Comparison `==` (type coercion), `===` (exact match), `!=`, `!==`, `>`, `<`, `>=`, `<=`.

```
let a = 10, b = 3;
let sum = a + b;        // 13
let mod = a % b;        // 1
let pow = a ** 2;       // 100
let eqLoose = '10' == 10;   // true (coerces string to number)
let eqStrict = '10' === 10; // false (types differ)
```

Listing 4: Operators Example

Explanation: Prefer `===` to avoid bugs from coercion (e.g., `[] == 0` is true). Modulo is useful for cycles; exponentiation for powers.

3.2 Conditional Statements

Conditionals execute code based on boolean expressions, enabling decision-making.

3.2.1 If-Else

The `if` statement evaluates a condition; if true, executes the block; else (optional) runs alternative code. Supports chaining with `else if`.

```
let age = 25;
if (age >= 18) {
    console.log('Adult: Can vote');
} else if (age >= 13) {
    console.log('Teen: Parental consent needed');
} else {
    console.log('Child: Restricted access');
}
// Output: Adult: Can vote
```

Listing 5: If-Else Example

Explanation: Conditions are truthy/falsy evaluated (falsy: false, 0, ", null, undefined, NaN). Nesting allows complex logic.

3.2.2 Switch

`switch` compares an expression against cases using strict equality; efficient for multiple discrete values. Includes `break` to prevent fall-through and `default` for unmatched cases.

```
let day = 'Monday';
switch (day) {
    case 'Monday':
    case 'Tuesday':
        console.log('Workday start');
        break;
    case 'Saturday':
    case 'Sunday':
        console.log('Weekend');
        break;
    default:
        console.log('Midweek');
}
// Output: Workday start
```

Listing 6: Switch Example

Explanation: Cases without `break` fall through to the next, allowing grouped behaviors. Use for enums or fixed options over long if-chains.

3.3 Loops

Loops repeat code execution, iterating over data or until a condition fails.

`for` Initializes counter, checks condition, updates; ideal for known iterations.

`while/do-while` Condition-checked before/after body; `do-while` guarantees one execution.

`for...of` Iterates iterable values (arrays, strings); ES6.

`for...in` Iterates enumerable properties (object keys); use cautiously as it includes inherited ones.

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

Listing 7: For Loop Example

Explanation: The loop runs while `i < 5`, incrementing `i`. Use `break` to exit early, `continue` to skip iterations.

```
let fruits = ['apple', 'banana', 'cherry'];
for (let fruit of fruits) {
    console.log(fruit);  // Iterates values
}
```

Listing 8: For...of Loop Example

Explanation: `for...of` destructures iterables, avoiding index management; contrasts with `for...in` which logs indices: 0, 1, 2.

4 Functions

Functions encapsulate reusable code, accepting inputs (parameters) and returning outputs. As first-class objects, they can be assigned to variables, passed as arguments, or returned from other functions.

4.1 Function Declarations and Expressions

Declarations are hoisted; expressions are not. Arrow functions (ES6) provide concise syntax and lexical `this` binding.

4.1.1 Function Declaration

Named, hoisted functions for classic definition.

```
function greet(name) {  // Parameter: name
    return `Hello, ${name}!`;  // Return value
}
console.log(greet('Brenda'));  // 'Hello, Brenda!'
```

Listing 9: Function Declaration

Explanation: Call by name with arguments. Hoisting allows calls before definition. Use for utilities.

4.1.2 Arrow Functions (ES6)

Concise syntax: `(params) => expression`. Implicit return for single expressions; no own `this`.

```
const greet = (name) => `Hello, ${name}!`;
console.log(greet('Brenda'));  // 'Hello, Brenda!'
const add = (a, b) => a + b;
console.log(add(2, 3));      // 5
```

Listing 10: Arrow Function Example

Explanation: Arrows simplify callbacks; `this` inherits from enclosing scope, fixing common binding issues in event handlers.

4.2 Parameters and Defaults

Parameters are placeholders; defaults (ES6) provide fallbacks. Rest parameters (`...args`) collect extras into an array.

```
function add(a, b = 0) {  // Default for b
    return a + b;
}
console.log(add(5));  // 5 (b=0)

```

```
function sum(...numbers) {  // Rest: collects into array
    return numbers.reduce((acc, n) => acc + n, 0);
}
console.log(sum(1, 2, 3));  // 6
```

Listing 11: Default and Rest Parameters

Explanation: Defaults avoid undefined checks. Rest enables variable arguments, mimicking Python's *args.

4.3 Higher-Order Functions

Functions that operate on other functions: accept them as args or return them, enabling composition (e.g., map, filter).

```
const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2);      // [2, 4, 6]
const evens = numbers.filter(n => n % 2 === 0);  // [2]
const sum = numbers.reduce((acc, n) => acc + n, 0);  // 6
```

Listing 12: Higher-Order Functions Example

Explanation: `map` transforms each element; `filter` selects based on predicate; `reduce` aggregates. Chainable for data pipelines.

5 Arrays and Objects

Arrays are ordered lists; objects are unordered collections. Both are objects under the hood, supporting methods for manipulation.

5.1 Arrays

Dynamic, zero-indexed lists with length property. Support push/pop, splice, and ES6 iterators.

Key methods:

- `push/pop`: Add/remove from end.
- `shift/unshift`: From start (inefficient for large arrays).
- `indexOf(value)`: Find position.

```
let fruits = ['apple', 'banana'];
fruits.push('cherry');           // ['apple', 'banana', 'cherry']
let last = fruits.pop();         // 'cherry', array now ['apple', 'banana']
console.log(fruits[1]);          // 'banana' (index 1)
let index = fruits.indexOf('apple');  // 0
```

Listing 13: Array Operations Example

Explanation: Indices start at 0. Use `length` for bounds: `fruits.length === 2`. Avoid direct assignment beyond length to prevent holes.

5.2 Objects

Unordered collections of properties (key-value pairs), where keys are strings/symbols. Support dot/bracket notation.

Key features:

- `this`: Refers to the object in methods.

- Methods: Functions as properties.
- `Object.keys(obj)`: Array of keys.

```
let person = {
    name: 'Brenda',
    age: 1,
    greet: function() { return `Hi, I'm ${this.name}`; }  // Method
};
console.log(person.greet());      // 'Hi, I'm Brenda'
person.age = 2;                   // Mutation
let keys = Object.keys(person);  // ['name', 'age', 'greet']
```

Listing 14: Object Example

Explanation: `this` binds to the caller. Bracket notation for dynamic keys: `person['name']`.

5.2.1 Destructuring (ES6)

Extract values into variables from arrays/objects, simplifying assignments.

```
let {name, age} = person;         // name='Brenda', age=2
let [first, second] = fruits;     // first='apple', second='banana'
let [x, ...rest] = [1, 2, 3, 4]; // x=1, rest=[2,3,4]
```

Listing 15: Destructuring Example

Explanation: Rest (`...`) collects remainder. Rename with `let {name: alias} = person;`. Great for function params.

6 DOM Manipulation

The Document Object Model (DOM) is a tree representation of HTML. JavaScript queries and modifies it for dynamic UIs.

6.1 Selecting Elements

Access via IDs, classes, tags, or CSS selectors.

```
let elemById = document.getElementById('myId');
let elemByClass = document.querySelector('.myClass');  // First match
let elemsByTag = document.querySelectorAll('p');       // NodeList
```

Listing 16: DOM Selection Example

Explanation: `querySelector` uses CSS syntax (e.g., '#id', '.class', 'tag'). NodeList is array-like; iterate with for...of.

6.2 Modifying Elements

Change properties, styles, or content; handle events for interactivity.

```
let button = document.querySelector('button');
button.textContent = 'Click Me!';  // Change text
button.style.backgroundColor = 'blue';  // Inline style

button.addEventListener('click', function(event) {
    event.target.style.color = 'red';  // this === event.target
    console.log('Clicked!');
});
```

Listing 17: DOM Modification and Events Example

Explanation: `addEventListener` binds handlers; `event` provides details. Remove with `removeEventListener`. Use `innerHTML` cautiously (XSS risk).

7 Asynchronous JavaScript

Asynchrony handles operations like I/O without blocking, using callbacks, promises, or async/await for cleaner code.

7.1 Callbacks

Functions passed as arguments, executed on completion. Prone to "callback hell" with nesting.

```
function fetchData(callback) {
    setTimeout(() => {
        callback('Data loaded');
    }, 1000);
}
fetchData(data => console.log(data));  // 'Data loaded' after 1s
```

Listing 18: Callback Example

Explanation: `setTimeout` simulates async. Callbacks fire post-task; error handling via additional params.

7.2 Promises

Objects representing eventual completion/failure (ES6). Chain with `then/catch`; resolve/reject inside constructor.

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        if (Math.random() > 0.5) {
            resolve('Success!');
        } else {
            reject('Failure!');
        }
    }, 1000);
});
promise
    .then(result => console.log(result))
    .catch(error => console.error(error));
```

Listing 19: Promise Example

Explanation: `then` handles success, `catch` errors. Static methods: `Promise.all([p1, p2])` waits for all.

7.3 Async/Await (ES8)

Syntactic sugar over promises; `async` marks functions, `await` pauses until resolution.

```
async function fetchData() {
    try {
        let response = await fetch('https://api.example.com/data');
        if (!response.ok) throw new Error('Network error');
        let data = await response.json();
        return data;
    } catch (error) {
        console.error('Error:', error.message);
    }
}
fetchData().then(data => console.log(data));
```

Listing 20: Async/Await Example

Explanation: `try/catch` wraps awaits. Sequential async reads like sync; use for APIs. Top-level await in modules (ES2022).

8 Error Handling

Gracefully manage runtime errors to prevent crashes, using try-catch-finally.

```
try {
    let result = riskyOperation();  // Assume throws
    if (!result) throw new Error('Invalid result');
} catch (error) {
    console.log('Caught:', error.message);  // 'Invalid result'
} finally {
    console.log('Cleanup always runs');
}
```

Listing 21: Try-Catch-Finally Example

Explanation: `try` attempts code; `catch` handles exceptions (Error objects with `message`, `stack`); `finally` executes regardless (e.g., close files).

Custom errors: `throw new Error('Message')`. Use `console.error` for logging.

9 Modern ES6+ Features

ES6 (ECMAScript 2015) revolutionized JS with classes, modules, and syntax sugar. Later versions added async/await, optional chaining.

9.1 Template Literals

Backtick-enclosed strings with embedded expressions via ${ }, multiline support.

```
let name = 'Brenda';
let msg = `Hello, ${name}!
Multi-line
text.`;
console.log(msg);  // Interpolates and preserves newlines
let calc = `Sum: ${2 + 3}`;  // 'Sum: 5'
```

Listing 22: Template Literals Example

Explanation: Replaces concatenation; tagged templates allow custom processing (e.g., for styling).

9.2 Spread and Rest Operators

... expands iterables (spread) or collects args (rest); versatile for arrays/objects.

```
let arr1 = [1, 2];
let arr2 = [...arr1, 3, ...[4, 5]];  // [1,2,3,4,5] (spread)
let obj1 = {a: 1};
let obj2 = {...obj1, b: 2};          // {a:1, b:2} (shallow copy)

```

```
function sum(...numbers) {           // Rest
    return numbers.reduce((a, b) => a + b, 0);
}
console.log(sum(1, 2, 3));           // 6
```

Listing 23: Spread and Rest Example

Explanation: Spread clones/merges; rest variadics. Object spread (ES2018) for immutability.

9.3 Modules

Organize code into files; `export/import` for dependency management (ES6).

```
// math.js
export const pi = 3.14;
export function add(a, b) { return a + b; }
export default function multiply(a, b) { return a * b; }  // Default

// main.js
import multiply, { pi, add } from './math.js';
console.log(add(2, 3) * pi);  // ~15.7
```

Listing 24: Modules Example

Explanation: Named exports for specifics, default for main. Use bundlers like Webpack for browser compatibility. Promotes reusability.

9.4 Classes (ES6)

Syntactic sugar over prototypes for OOP: constructors, methods, inheritance.

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log(`${this.name} makes a sound`);
    }
}
class Dog extends Animal {
    speak() {
        super.speak();  // Call parent
        console.log('Woof!');
    }
}
let dog = new Dog('Rex');
dog.speak();  // 'Rex makes a sound' then 'Woof!'
```

Listing 25: Classes Example

Explanation: `extends/super` for inheritance. Static methods on class: `Animal.info()`. Private fields (#prop) in ES2022.

9.5 Optional Chaining and Nullish Coalescing (ES2020)

Safe navigation: `?.` avoids errors on undefined/null. `??` defaults only to null/undefined (not falsy like 0).

```
let user = { profile: { name: 'Brenda' } };
console.log(user?.profile?.name);     // 'Brenda'
console.log(user?.settings?.theme ?? 'light');  // 'light' (if undefined)
```

Listing 26: Optional Chaining Example

Explanation: Prevents `TypeError`; chaining like `arr?.[0]?.prop`. Nullish distinguishes from falsy.

10 Conclusion

These lectures provide a comprehensive foundation in JavaScript, from basics to modern features. Each concept—variables, strings, functions, ES6 syntax like arrows and destructuring—builds toward robust applications. Practice by building a to-do app with DOM/events or an API fetcher with async/await.

Further reading: MDN Web Docs, ”Eloquent JavaScript” book. Explore frameworks like React for UI, Node.js for backend.