JavaScript Tutorial for Beginners

Complete Course Notes — Learn JavaScript in 2 Hours

SECTION 1: Introduction to JavaScript

1.1 What is JavaScript?

JavaScript is the world's most popular front-end programming language. It works alongside HTML and CSS to create fully interactive web pages.

Layer	Role
HTML	Defines the structure and content of the page (headings, paragraphs, buttons, forms)
CSS	Controls visual styling — colors, fonts, layout, spacing
JavaScript	Adds behavior and interactivity — responding to user actions, updating content dynamically, performing calculations

- JavaScript makes web pages interactive — show/hide menus, validate forms, run calculators, fetch live data
- Example: Google's search suggestions appearing as you type are powered by JavaScript
- JavaScript is case-sensitive — uppercase and lowercase letters are treated differently
- JavaScript is an interpreted language — each line is executed one at a time by the browser, with no prior compilation needed

1.2 Where Does JavaScript Run?

- Inside the browser — Chrome, Firefox, Edge, Safari, Opera all include a built-in JavaScript engine
- Outside the browser — using Node.js, JavaScript can run directly on a server or any machine with a JavaScript interpreter
- Node.js internally uses the V8 engine (developed by Google) — the same engine inside Chrome
- Node.js command to run a file: node filename.js

1.3 Three Ways to Execute JavaScript

Method	When to Use It
Internal JavaScript	JavaScript written inside a <script> tag in the same HTML file. Good for single-page scripts that don't need to be shared.
External JavaScript	JavaScript written in a separate .js file, linked to one or more HTML pages. Best practice for larger apps — allows code reuse across multiple pages.
Node.js (command prompt)	Run JavaScript directly on the computer without a browser. Ideal for practicing core language features (functions, arrays, objects) and for server-side programming.

1.4 JavaScript History and Versions

- Created by Brendan Eich at Netscape Corporation in 1995; first formally released as ES1 in 1997
- ECMAScript (ES) is the official specification published by ECMA International — JavaScript is its practical browser implementation
- ES5 (2009) — introduced strict mode, JSON support, important array and object methods
- ES6 / ES2015 — landmark update: introduced let, const, classes, arrow functions, template literals, de structuring, modules
- Versions from ES2015 onward are named by year (ES2016, ES2017 … ES2020+)
- All version numbers of ECMAScript and JavaScript are identical — ES6 = JavaScript 2015
- File extension for JavaScript files: .js

SECTION 2: Variables and Data Types

2.1 What is a Variable?

A variable is a named memory location in RAM used to store a value temporarily while a program is running. Variables hold inputs, intermediate results, and outputs.

- JavaScript variables are dynamically typed — you do not declare a fixed data type
- The same variable can hold a number today and a string tomorrow
- Variable names: can include letters, digits, underscore _, dollar $ — cannot contain spaces or most special characters
- Naming convention: camelCase is strongly recommended — e.g. securedMarks, firstName, numberOfYears
- Default value of any uninitialized variable is undefined

2.2 Declaring Variables — var, let, const

Keyword	Scope / Behavior
var	Function-scoped (or global if outside a function). Declarations are hoisted to the top of their scope. Variables declared with var inside a block (if, for, while) leak out and remain accessible after the block ends.
let (recommended)	Block-scoped — the variable only exists inside the { } block where it is declared. Not hoisted. Introduced in ES6/ES2015. Replaces var for most use cases.
const	Block-scoped like let, but the value cannot be reassigned after declaration. Use for values that should not change.

Example: declaring and initializing variables

```
let name = "Scott";
let age = 20;
let isPass = true;
let score;           // default value is undefined
```

2.3 Data Types in JavaScript

Data Type	Example
Number	42, 3.14, -7
String	"hello", 'world'
Boolean	true, false
undefined	(unassigned variable)
Object	{ name: "Scott", age: 20 }
Array	[1, 2, 3]
Function	function() { ... }

Note: The table above uses three columns. In JavaScript objects, arrays, and functions can all be stored inside variables just like numbers and strings.

2.4 Output — console.log()

- console.log() is a built-in function that prints a value to the console/command prompt
- Pass any value — number, string, variable name, or expression — inside the parentheses
- Every JavaScript statement should end with a semicolon ; (optional but strongly recommended)

```
console.log(1);           // output: 1
console.log("hello");     // output: hello
let x = 42;
console.log(x);           // output: 42
```

2.5 Comments

- Single-line comment: double slash //
- Multi-line comment: /* ... */

```
// This is a single-line comment
/* This is a
   multi-line comment */
```

SECTION 3: Operators

3.1 Overview of JavaScript Operators

An operator is a symbol that performs a specific operation on one or more values. JavaScript has six main categories of operators.

3.2 Arithmetic Operators

Operator	Operation — Example → Result
+	Addition — 5 + 3 → 8
-	Subtraction — 5 - 3 → 2
*****	Multiplication — 5 * 3 → 15
/	Division — 10 / 2 → 5
%	Modulus (remainder) — 10 % 3 → 1

3.3 Assignment Operators

Operator	Meaning
=	Assign: x = 5 stores 5 in x
+=	Add then assign: x += 3 is shorthand for x = x + 3
-=	Subtract then assign
***=**	Multiply then assign
/=	Divide then assign
%=	Modulus then assign

3.4 Increment / Decrement Operators

- ++ (increment) — increases the value of a variable by 1
- -- (decrement) — decreases the value of a variable by 1
- Prefix (++x): increments first, then returns the new value
- Postfix (x++): returns the current value first, then increments

3.5 Relational (Comparison) Operators

Operator	Meaning
==	Equal to (value only — loose comparison)
===	Strictly equal (value AND type)
!=	Not equal to
!==	Strictly not equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

3.6 Logical Operators

Operator	Name
&&	AND
||	OR
!	NOT

3.7 Concatenation Operator

- The + operator doubles as a string concatenation operator when used with strings
- Joining two strings: "Hello" + " World" → "Hello World"
- Joining string + number: "Age: " + 20 → "Age: 20" (number is converted to string)

SECTION 4: Conditional Statements — if, if-else

4.1 Simple if Statement

The simple if statement executes a block of code only when a condition is true. If the condition is false, the block is skipped entirely.

```
if (condition) {
  // block executes only if condition is true
}
```

- A code block is enclosed in curly braces { }
- One or more statements can go inside the block
- If the condition is false, execution jumps to the statement after the closing brace

Example — exam result:

```
let securedMarks = 60;
let minPass = 35;
if (securedMarks < minPass) {
  console.log("Fail");
}
if (securedMarks >= minPass) {
  console.log("Pass");
}
```

4.2 if-else Statement

if-else handles both outcomes: when the condition is true the true block runs; when false the else block runs. Exactly one block always executes.

```
if (condition) {
  // runs when condition is true
} else {
  // runs when condition is false
}
```

Statement Type	Blocks
Simple if	1 block
if-else	2 blocks (true + false)

- Only one else is allowed per if statement
- Use if-else when you have two mutually exclusive outcomes — e.g. pass/fail, logged in/not logged in, open/closed

SECTION 5: Loops — while and for

5.1 Why Use Loops?

Loops execute a block of code repeatedly as long as a condition remains true. Without loops, repeating an action 100 times would require writing 100 identical lines of code.

5.2 while Loop

The while loop checks the condition before each iteration. Use it when you do NOT know in advance how many times you need to loop.

```
let i = 1;
while (i <= 5) {
  console.log(i);
  i++;
}
```

- Execution sequence: initialise → check condition → execute block → increment → check condition → repeat
- Always include an increment/decrement inside the loop — forgetting it creates an infinite loop that never ends
- Use while when: looping until a user quits, reading data from a source until it ends, checking a live value like a stock price

5.3 for Loop

The for loop puts initialisation, condition, and increment all on one line. Use it when you know exactly how many iterations are needed.

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

- All three loop controls are in one place — easier to read and less likely to create infinite loops
- Execution sequence: initialise (once) → condition → block → increment → condition → block → repeat
- Decremented loop (counting down): for (let i = 5; i >= 1; i--)
- Use for when: printing numbers 1–100, processing a fixed number of students, repeating exactly N times

5.4 while vs for — Comparison

Feature	while Loop
Best when	Number of iterations unknown — depends on external input or condition
Loop controls location	Initialisation above loop; increment inside loop body
Readability	Can be harder to track when loop ends
Example use case	Keep running while user keeps providing valid input

5.5 Debugging Loops in VS Code

- Add a breakpoint: click in the left margin next to a line — a red dot appears
- Start debugging: Run menu → Start Debugging (F5) → choose Node.js
- Step over (F10): execute the current line and move to the next
- Step into (F11): if the current line calls a function, enter that function's definition
- The Variables panel (top-left) shows the current value of each variable after every step
- Remove breakpoint: click the red dot again to remove it
- Stop: click the red square stop button in the debug toolbar

SECTION 6: var vs let — Scope and Hoisting

6.1 Scope Comparison

Scope defines where a variable is accessible within the code. The key difference between var and let is their scope behavior inside blocks.

Behavior	var
Global scope (outside any function or block)	Creates a global variable — accessible everywhere
Local scope (inside a function)	Creates a function-scoped local variable
Block scope (inside if, for, while, etc.)	Leaks out — the variable is accessible OUTSIDE the block after it closes
Hoisting	Declaration is hoisted to the top of its scope — can be referenced before the line where it is declared
ES version introduced	Original JavaScript
Recommended?	No — use let instead

6.2 Block Scope Example

```
if (true) {
  var x = 10;    // leaks out — accessible after this block
  let y = 20;    // block-scoped — NOT accessible after this block
}
console.log(x);  // 10  (var leaks out)
console.log(y);  // ReferenceError: y is not defined
```

- The let keyword is the only way to declare a truly block-scoped variable in JavaScript
- Use let inside for loops — the loop variable (i) is destroyed when the loop ends and does not pollute the outer scope
- Rule of thumb: always use let (or const) — avoid var in modern JavaScript

SECTION 7: Functions

7.1 What is a Function?

A function is a named, reusable block of statements that performs a specific task. It can accept inputs (parameters) and return an output (return value).

- A function defined but never called will never execute
- The same function can be called many times — each call re-executes the body
- Benefits: reusability (write once, call many times) and modularity (break large programs into smaller tasks)

7.2 Function Anatomy

```
function functionName(param1, param2) {
  // statements
  return result;
}
```

Term	Meaning
function keyword	Tells JavaScript you are defining a function
functionName	The identifier used to call the function — choose a descriptive name based on purpose
Parameters / Arguments	Input values the function receives. Parameters are the names in the definition; arguments are the actual values passed when calling.
return statement	Sends a value back to the caller. Execution of the function stops at return.
return value	The value sent back. If no return statement exists, the function returns undefined by default.

7.3 Two Ways to Define a Function

Method 1 — Function Declaration:

```
function showCity() {
  console.log("New York");
}
showCity();   // call the function
```

Method 2 — Function Expression (storing in a variable):

```
let getSimpleInterest = function(principal, rate, years) {
  let si = (principal * rate * years) / 100;
  return si;
};
console.log(getSimpleInterest(1000, 6.7, 3));  // 201
```

- Both approaches work — function expressions make it explicit that functions are values (first-class citizens)
- Call a function by writing its name followed by parentheses: functionName(arg1, arg2)
- Store the return value: let result = getSimpleInterest(1000, 6.7, 3);

7.4 Functions with No Arguments and No Return

The simplest functions perform a task without receiving inputs or returning a value:

```
function showCity() {
  console.log("New York");
}
showCity();   // New York
showCity();   // New York (called again — same output)
```

- Every time the function is called, execution jumps to the function body, runs all statements, then returns to the call site
- If a function has no return statement, JavaScript automatically returns undefined

7.5 Execution Flow Summary

Step	What Happens
1 — Define function	JavaScript creates the function in memory. Nothing inside the body executes yet.
2 — Call function	Execution jumps from the call site into the function body.
3 — Execute body	Statements run top to bottom. Parameters receive the argument values.
4 — return statement	The return value is sent back to the call site. Function execution ends.
5 — Continue	Execution resumes at the call site with the returned value available.

SECTION 8: Objects

8.1 What is an Object?

An object is a memory unit that groups related data (properties) and behaviors (methods) together. Everything physical or conceptual can be modelled as an object.

Concept	Explanation
Property	A key-value pair where the value is a simple data type (number, string, Boolean). Represents a characteristic — e.g. carColor: "red"
Method	A key-value pair where the value is a function. Represents a behavior — e.g. start: function() { ... }
Key	The name part of a key-value pair — used to access the property or method
Value	The data stored under the key
Object Literal	The syntax { key: value, key: value } used to create an object directly in code
Reference Variable	A variable that stores a reference (pointer) to an object in memory

8.2 Creating an Object — Object Literal Syntax

```
let person = {
  personName: "Scott",
  age: 20,
  birthday: function() {
    return "Happy Birthday!";
  }
};
```

- Use a colon : (not =) to assign values to keys inside an object
- Separate key-value pairs with commas
- Curly braces { } define the object — do not use = inside
- An empty object is valid: let obj = {};

8.3 Accessing Properties and Methods

```
console.log(person.personName);    // Scott
console.log(person.age);           // 20
console.log(person.birthday());    // Happy Birthday!
```

- Dot notation: variableName.keyName — the most common way to access object members
- Bracket notation: variableName["keyName"] — useful when the key name is stored in a variable
- To call a method, include parentheses: person.birthday() — without parentheses you get the function reference, not its return value
- Always store the object in a reference variable — objects are nameless in memory and can only be accessed through their variable

8.4 JavaScript as an Object-Oriented Language

- JavaScript is fundamentally object-oriented — it uses objects to model real-world entities
- Objects group related data and operations together — this is the core idea of OOP
- Classes were introduced in ES6 but objects and OOP principles existed in JavaScript from the beginning
- Methods manipulate properties — e.g. a birthday() method increments the age property
- In large applications, objects represent entities like users, products, orders, and server requests

— End of Course Notes —