Complete JS Course Syllabus

📦 Chapter 1: Variables & Declarations

(JavaScript – Learn Everything Series by Sheryians Coding School)

🧠 What are Variables?

Variables are containers that hold data.
They help us store, reuse, and update information in JavaScript — from simple values like numbers to complex data like arrays and objects.

Think of a variable as a box with a name on it. You can put something inside it (a value), and later check or change what's inside.
In JavaScript, you create these boxes using keywords: `var`, `let`, or `const`.

🧪 var, let, and const – Line-by-Line Comparison

👴 `var` – Old and risky

- Scoped to **functions**, not blocks
- Can be **redeclared** and **reassigned**
- **Hoisted** to the top with `undefined` value

```js
var score = 10;
var score = 20; // OK
```

👨‍🏫 `let` – Modern and safe

- Scoped to **blocks** (`{}`)

- Can be **reassigned** but **not redeclared**
- Hoisted, but stays in the **Temporal Dead Zone (TDZ)**

```js
let age = 25;
age = 30;      // ✅
let age = 40;  // ❌ Error (same block)
```

🔐 `const` – Constant values

- Scoped to **blocks**
- **Cannot be reassigned** or **redeclared**
- Value must be assigned at declaration
- TDZ applies here too

```js
const PI = 3.14;
PI = 3.14159; // ❌ Error
```

👉 But: If `const` holds an object/array, you can still change its contents:

```js
const student = { name: "Riya" };
student.name = "Priya";  // ✅ OK
student = {};            // ❌ Error
```

🔥 Scope in Real Life

- **Block Scope** → Code inside `{}` like in loops, `if`, etc.
- **Function Scope** → Code inside a function
- `let` and `const` follow block scope.
- `var` ignores block scope — which leads to bugs.

```js
```

```
{
  var x = 5;
  let y = 10;
  const z = 15;
}
console.log(x); // ✅ 5
console.log(y); // ❌ ReferenceError
console.log(z); // ❌ ReferenceError
```

🧨 Hoisting

JavaScript prepares memory before running code.

It moves all declarations to the top — this is called **hoisting**.

But:

- `var` is hoisted and set to `undefined`
- `let` and `const` are hoisted but not initialized — so accessing them early gives **ReferenceError**

js

```
console.log(a); // undefined
var a = 10;
```

js

```
console.log(b); // ❌ ReferenceError
let b = 20;
```

⚠️ Common Confusions (JS Reality Checks)

- `const` doesn't make things *fully* constant. It protects the *variable*, not the *value*.
- `var` is outdated — it's better to use `let` and `const`.

- `let` and `const` behave similarly, but `const` gives more safety — use it when you're not planning to reassign.

🧠 Mindset Rule

> Use `const` by default. Use `let` only when you plan to change the value.
> Avoid `var` — it belongs to the past.

🧪 Practice Zone

1. Declare your name and city using `const`, and your age using `let`.
2. Try this and observe the result:

```js
let x = 5;
let x = 10;
```

3. Guess the output:

```js
console.log(count);
var count = 42;
```

4. Create a `const` object and add a new key to it — does it work?
5. Try accessing a `let` variable before declaring it — what error do you see?
6. Change a `const` array by pushing a value. Will it throw an error?

🧠 Chapter 2: Data Types + Type System

(JavaScript – Learn Everything Series by Sheryians Coding School)

📦 What Are Data Types?

In JavaScript, every value has a type.
These types define what kind of data is being stored — a number, text, boolean, object, etc.

There are two categories:

- **Primitive types** – stored directly.
- **Reference types** – stored as memory references.

🔷 Primitive Data Types

1. **String** → Text
   `"hello"`, `'Sheryians'`
2. **Number** → Any numeric value
   `3`, `-99`, `3.14`
3. **Boolean** → True or false
   `true`, `false`
4. **Undefined** → Variable declared but not assigned
   `let x;` → `x` is `undefined`
5. **Null** → Intentional empty value
   `let x = null;`
6. **Symbol** → Unique identifier (rarely used)
7. **BigInt** → Very large integers
   `1234567890123456789012345678 90n`

🔷 Reference Data Types

- **Object** → `{ name: "Harsh", age: 26 }`

- **Array** → `[10, 20, 30]`
- **Function** → `function greet() {}`

These are not copied directly, but by reference.

🔍 typeof Operator

Used to check the data type of a value:

```js
typeof "Sheryians"      // "string"
typeof 99               // "number"
typeof true             // "boolean"
typeof undefined        // "undefined"
typeof null             // "object" ← known bug
typeof []               // "object"
typeof {}               // "object"
typeof function(){}     // "function"
```

Note: `typeof null === "object"` is a bug, but has existed since the early days of JS.

🔁 Type Coercion (Auto-Conversion)

JavaScript auto-converts types in some operations:

```js
"5" + 1     // "51"   → number converted to string
"5" - 1     // 4      → string converted to number
true + 1    // 2
null + 1    // 1
undefined + 1 // NaN
```

🚨 Loose vs Strict Equality

- `==` compares **value** with type conversion
- `===` compares **value + type** (no conversion)

```js
5 == "5"    // true
5 === "5"   // false
```

Always prefer `===` for accurate comparisons.

🧪 NaN – Not a Number

```js
typeof NaN      // "number"
```

Even though it means *"Not a Number"*, NaN is actually of type `number`.
This is because operations like `0 / 0` or `parseInt("abc")` still produce a numeric result — just an invalid one.

🔦 Truthy and Falsy Values

Falsy values:
`false`, `0`, `""`, `null`, `undefined`, `NaN`

Everything else is **truthy**, including:
`"0"`, `"false"`, `[]`, `{}`, `function(){}`

Example:

```js
if ("0") {
  console.log("Runs");  // "0" is a non-empty string = truthy
}
```

🧠 Mindset

JavaScript will often **auto-convert** types behind the scenes.
Always stay aware of what data type you're working with.

❓ Common Confusions

- `typeof null` is `"object"` — this is a bug.
- `undefined` means the variable was never assigned.
  `null` means you intentionally set it to "nothing".
- `'5' + 1` is `"51"` but `'5' - 1` is `4`.

🧪 Practice Zone

1. Predict the output:

```js
console.log(null + 1);
console.log("5" + 3);
console.log("5" - 3);
console.log(true + false);
```

2. Check types:

```js
console.log(typeof []);
console.log(typeof null);
console.log(typeof 123n);
```

3. Truthy or Falsy?

```js
```

```
console.log(Boolean(0));        // falsy
console.log(Boolean("0"));      // truthy
console.log(Boolean([]));       // truthy
console.log(Boolean(undefined));// falsy
```

4. Write a function `isEmpty(value)` that checks if a given value is `null`, `undefined`, or `""`.
5. Compare with loose vs strict:

```
js

console.log(5 == "5");   // ?
console.log(5 === "5");  // ?
```

🔄 Chapter 3: Operators

(JavaScript – Learn Everything Series by Sheryians Coding School)

🔧 What are Operators?

Operators are special symbols or keywords in JavaScript used to perform operations on values (operands).
You'll use them in calculations, comparisons, logic, assignments, and even type checks.

Think of them as the verbs of your code — they **act on data**.

➕ Arithmetic Operators

Used for basic math.

```
js

+   // addition
-   // subtraction
*   // multiplication
```

```
/   // division
%   // modulus (remainder)
**  // exponentiation (power)
```

Example:

```js
let a = 10, b = 3;
console.log(a + b);   // 13
console.log(a % b);   // 1
console.log(2 ** 3);  // 8
```

🧱 Assignment Operators

Assign values to variables.

```
=       // assigns value
+=      // a += b  => a = a + b
-=      // a -= b
*=, /=, %=
```

Example:

```js
let score = 5;
score += 2;     // score = 7
```

🧾 Comparison Operators

Used in condition checks.

```
js
```

```
==      // equal (loose)
===     // equal (strict — value + type)
!=      // not equal (loose)
!==     // not equal (strict)
> < >= <=
```

Example:

```js
console.log(5 == "5");    // true
console.log(5 === "5");   // false
```

✅ Logical Operators

Used to combine multiple conditions.

```
&&  // AND — both must be true
||  // OR — either one true
!   // NOT — negates truthiness
```

Example:

```js
let age = 20, hasID = true;
if (age >= 18 && hasID) {
  console.log("Allowed");
}
```

🌀 Unary Operators

Used on a single operand.

```
```

```
+   // tries to convert to number
-   // negates
++  // increment
--  // decrement
typeof  // returns data type
```

Example:

```js
let x = "5";
console.log(+x);  // 5 (converted to number)
```

❓ Ternary Operator (Conditional)

Shorthand for `if...else`

```js
condition ? valueIfTrue : valueIfFalse
```

Example:

```js
let score = 80;
let grade = score > 50 ? "Pass" : "Fail";
```

🧪 typeof Operator

```js
typeof 123        // "number"
typeof "hi"       // "string"
typeof null       // "object" (JS bug)
typeof []         // "object"
```

```
typeof {}          // "object"
typeof function(){} // "function"
```

🧠 Mindset

Operators make logic happen.

They help you **make decisions**, **combine values**, and **create expressions**.

Try to:

- Use `===` instead of `==` to avoid type bugs.
- Use ternary for quick decisions, not complex ones.
- Think in truthy/falsy when using `&&`, `||`, `!`.

❓ Common Confusions

- `"5" + 1` is `"51"` (string concat), but `"5" - 1` is `4` (number subtract)
- `!!value` is a quick trick to convert anything into a boolean
- Pre-increment (`++i`) vs post-increment (`i++`) return different results

---

🧪 Practice Zone

1. Predict:

```js
console.log("10" + 1);
console.log("10" - 1);
console.log(true + false);
console.log(!!"Sheryians");
```

2. Convert using unary `+`

```js
```

```
let str = "42";
let num = +str;
console.log(num);  // 42
```

3. Use ternary:

```
js

let age = 17;
let msg = age >= 18 ? "Adult" : "Minor";
```

4. Build a calculator:

```
js

// Using switch + arithmetic operators
function calc(a, b, operator) {
  // +, -, *, /
}
```

5. Score logic:

```
js

let marks = 82;
// Print "Excellent", "Good", "Average", or "Fail" based on ranges
```

🧭 Chapter 4: Control Flow

(JavaScript – Learn Everything Series by Sheryians Coding School)

🚦 What is Control Flow?

Control flow decides **which code runs**, **when it runs**, and **how many times it runs**.
It's like decision-making + direction in your JavaScript program.

If operators are the verbs, **control flow is the traffic signal**.

🪜 if, else if, else

```js
if (condition) {
  // runs if condition is true
} else if (anotherCondition) {
  // runs if first was false, second is true
} else {
  // runs if none are true
}
```

✅ Example:

```js
let marks = 78;

if (marks >= 90) {
  console.log("A");
} else if (marks >= 75) {
  console.log("B");
} else {
  console.log("C");
}
```

🌀 switch-case

Great for checking **one variable against many values**.

```js
```

```
switch (value) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // fallback
}
```

✅ Example:

```js
let fruit = "apple";

switch (fruit) {
  case "banana":
    console.log("Yellow");
    break;
  case "apple":
    console.log("Red");
    break;
  default:
    console.log("Unknown");
}
```

🔁 Early Return Pattern

Used in functions to **exit early** if some condition fails.

```js
function checkAge(age) {
  if (age < 18) return "Denied";
  return "Allowed";
}
```

This avoids deep nesting and makes logic cleaner.

⚠️ Common Confusions

- `switch-case` executes **all cases** after a match unless you `break`
- `else if` chain works top-down — order matters
- You can use **truthy/falsy** values directly in `if`

🧠 Mindset

Control flow = **conditional storytelling**.
It helps your program make choices and **respond differently** to different inputs.

Write readable branches. Avoid nesting too deep — use early return if needed.

🧪 Practice Zone

1. Student grade logic:

```js
// Write a program that prints A, B, C, D, or F based on marks
```

2. Rock-paper-scissors:

```js
// Given player1 and player2's choice, print winner or draw
```

3. Login message:

```js

```

```
let isLoggedIn = true;
let isAdmin = false;

// Show different messages based on combination
```

4. Weather advice:

```
js

let weather = "rainy";

// Use switch-case to print what to wear
```

5. Age checker:

```
js

// Return "Kid", "Teen", "Adult", or "Senior"
```

🔁 Chapter 5: Loops

(JavaScript – Learn Everything Series by Sheryians Coding School)

🔁 Why Loops?

Loops help us **repeat code** without rewriting it.

If a task needs to be done **multiple times** (e.g., printing 1–10, going through an array, or checking each character in a string), loops are the **backbone**.

🔁 `for` Loop

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

- Start from `i = 0`
- Run till `i < 5`
- Increase `i` each time

🔁 `while` Loop

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

- Condition is checked **before** running

🔁 `do-while` Loop

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

- Runs **at least once**, even if condition is false

⛔ `break` and `continue`

- `break`: Exit loop completely
- `continue`: Skip current iteration and move to next

```js
for (let i = 1; i <= 5; i++) {
  if (i === 3) continue;
  console.log(i);  // Skips 3
}
```

🌀 `for-of` – Arrays & Strings

```js
for (let char of "Sheryians") {
  console.log(char);
}
```

- Works on anything iterable (arrays, strings)

📦 `forEach` – Arrays

```js
let nums = [10, 20, 30];
nums.forEach((num) => {
  console.log(num);
});
```

- Cleaner than `for` for arrays, but **you can't break/return**

🧱 `for-in` – Objects (and arrays if needed)

```js
let user = { name: "Harsh", age: 26 };
for (let key in user) {
  console.log(key, user[key]);
}
```

- Goes over **keys** in objects

⚠️ Common Confusions

- `for-in` is for objects, not arrays (may cause issues with unexpected keys)
- `forEach` can't use `break` or `continue`
- `while` and `do-while` work best when number of iterations is unknown

🧠 Mindset

Loops = **data processor**.
Use the **right loop** for the job:

- `for` = best for numbers/indexes
- `for-of` = for array values
- `for-in` = for object keys
- `while` = for unpredictable conditions

🧪 Practice Zone

1. **Print 1 to 10 using `for`**
2. **Print even numbers between 1 to 20**
3. **Reverse a string using loop**

4. **Sum of all numbers in an array**
5. **Print all characters of a name using `for-of`**
6. **Print all object keys and values using `for-in`**
7. **Use `continue` to skip a specific number**
8. **Guess number game – use `while` to ask until correct**
9. **Pattern: Print triangle using `*`**
10. **Sum of even numbers in an array using `forEach`**

🧮 Chapter 6: Functions

(JavaScript – Learn Everything Series by Sheryians Coding School)

🧠 What are Functions?

Functions are **blocks of reusable logic**.
Instead of repeating the same task again and again, wrap it in a function and **reuse it with different inputs**.

Think of a function like a vending machine:

- Input: you give money + item code
- Output: it gives you the item
- Logic: hidden inside

🛠️ Function Declarations

```
js
```

```
function greet() {
  console.log("Welcome to Sheryians!");
}
greet();
```

You **define** it once, then **call** it whenever needed.

🧾 Parameters vs Arguments

```
function greet(name) {
  console.log("Hello " + name);
}
greet("Harsh");
```

- `name` is a **parameter**
- `"Harsh"` is the **argument** you pass

🌀 Return Values

```
function sum(a, b) {
  return a + b;
}
let total = sum(5, 10); // total is 15
```

- `return` sends back a result to wherever the function was called
- After `return`, function exits

🧰 Function Expressions

```js
const greet = function () {
  console.log("Hello!");
};
```

- Functions stored in variables
- Cannot be hoisted (you can't call them before they're defined)

🏹 Arrow Functions

```js
const greet = () => {
  console.log("Hi!");
};
```

- Cleaner syntax
- No own `this`, no `arguments` object

🧂 Default + Rest + Spread

```js
function multiply(a = 1, b = 1) {
  return a * b;
}

function sum(...nums) {
  return nums.reduce((acc, val) => acc + val, 0);
}

let nums = [1, 2, 3];
console.log(sum(...nums)); // Spread
```

- `a = 1` → default parameter
- `...nums` → rest parameter
- `...nums` (in call) → spread operator

🎯 First-Class Functions

JavaScript treats functions as **values**:

- Assign to variables
- Pass as arguments
- Return from other functions

```js
function shout(msg) {
  return msg.toUpperCase();
}
function processMessage(fn) {
  console.log(fn("hello"));
}
processMessage(shout);
```

🧠 Higher-Order Functions (HOF)

Functions that **accept other functions** or **return functions**.

```js
function createMultiplier(x) {
  return function (y) {
    return x * y;
  };
}
let double = createMultiplier(2);
console.log(double(5)); // 10
```

🔐 Closures & Lexical Scope

Closures = when a function **remembers its parent scope**, even after the parent has finished.

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

Even after `outer` is done, `counter` still remembers `count`.

⚡ IIFE – Immediately Invoked Function Expression

```js
(function () {
  console.log("Runs immediately");
})();
```

Used to **create private scope instantly**.

🚀 Hoisting: Declarations vs Expressions

```js
hello(); // works
function hello() {
  console.log("Hi");
}
```

```
greet(); // error
const greet = function () {
  console.log("Hi");
};
```

- Declarations are hoisted
- Expressions are not

⚠️ Common Confusions

- Arrow functions don't have their own `this`
- You **can't break out** of `forEach`
- Closures often trap old variable values
- Return vs console.log – don't mix them up

🧠 Mindset

Functions are your **logic blocks + memory holders (via closure)**.
They keep your code **clean, DRY**, and **reusable**.

🧪 Practice Zone

1. **Write a BMI calculator function**
2. **Create a greet function with default name**
3. **Sum all numbers using rest parameter**
4. **Create a closure counter function**
5. **Write a function that returns another function**
6. **Use a function to log even numbers in array**
7. **Create a pure function to add tax**

8. **Use IIFE to show welcome message**
9. **Write a discount calculator (HOF style)**
10. **Make a toUpperCase transformer using HOF**

🧰 Chapter 7: Arrays

(JavaScript – Learn Everything Series by Sheryians Coding School)

🧠 What is an Array?

An array is like a **row of boxes**, where each box holds a value and has an index (0, 1, 2...). Arrays help you **store multiple values** in a single variable — numbers, strings, or even objects/functions.

```js
let fruits = ["apple", "banana", "mango"];
```

🏗️ Creating & Accessing Arrays

```js
let marks = [90, 85, 78];
console.log(marks[1]); // 85
marks[2] = 80;         // Update index 2
```

- Indexing starts from 0
- You can access, update, or overwrite values by index

⚙️ Common Array Methods

🧰 Modifiers (Change original array)

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

arr.push(5);      // Add to end
arr.pop();        // Remove last

arr.shift();      // Remove first
arr.unshift(0);   // Add to start

arr.splice(1, 2); // Remove 2 items starting at index 1
arr.reverse();    // Reverse order
```

🔍 Extractors (Don't modify original array)

```js
let newArr = arr.slice(1, 3); // Copy from index 1 to 2

arr.sort();  // Lexical sort by default
```

🔄 Iteration Methods

map()

Returns a new array with modified values.

```js
let prices = [100, 200, 300];
let taxed = prices.map(p => p * 1.18);
```

filter()

Filters out elements based on a condition.

```js
let nums = [1, 2, 3, 4];
let even = nums.filter(n => n % 2 === 0);
```

reduce()

Reduces the array to a single value.

```js
let total = nums.reduce((acc, val) => acc + val, 0);
```

forEach()

Performs an action for each element (but returns undefined).

```js
nums.forEach(n => console.log(n));
```

---

find(), some(), every()

```js
nums.find(n => n > 2);       // First match
nums.some(n => n > 5);       // At least one true
nums.every(n => n > 0);      // All true
```

✂️ Destructuring & Spread

```js
let [first, second] = ["a", "b", "c"];
let newArr = [...nums, 99]; // Spread to copy & add
```

⚠️ Common Confusions

- `splice` **changes** original array, `slice` **does not**
- `forEach` vs `map`: `map` returns a new array
- `sort()` converts values to strings unless `compareFn` is provided:

```js
[10, 2, 3].sort(); // [10, 2, 3] → ["10", "2", "3"] → wrong order
```

Use:

```js
arr.sort((a, b) => a - b); // Correct numeric sort
```

🧠 Mindset

Arrays are **structured, transformable data**.
You loop over them, transform them, filter them, or reduce them — all to control what shows up in your UI or logic.

🧪 Practice Zone

1. **Create an array of student names and print each**
2. **Filter even numbers from an array**
3. **Map prices to include GST (18%)**
4. **Reduce salaries to calculate total payroll**
5. **Find the first student with grade A**
6. **Write a function to reverse an array**
7. **Sort array of ages in ascending order**

8. **Destructure first two elements of an array**
9. **Use `some()` to check if any student failed**
10. **Use spread to copy and add new item**

📦 Chapter 8: Objects

(JavaScript – Learn Everything Series by Sheryians Coding School)

🧠 What is an Object?

Objects in JavaScript are like **real-world records** – a collection of **key-value pairs**.
They help us store **structured data** (like a student, a product, or a user profile).

```js
let student = {
  name: "Ravi",
  age: 21,
  isEnrolled: true
};
```

🔑 Key-Value Structure

- Keys are always strings (even if you write them as numbers or identifiers)
- Values can be anything – string, number, array, object, function, etc.

```js
console.log(student["name"]); // Ravi
console.log(student.age);     // 21
```

📍 Dot vs Bracket Notation

- Use **dot notation** for fixed key names
- Use **bracket notation** for dynamic or multi-word keys

```js
student["full name"] = "Ravi Kumar";  // ✅
student.course = "JavaScript";        // ✅
```

🏗️ Nesting and Deep Access

Objects can have **nested objects** (objects inside objects)

```js
let user = {
  name: "Amit",
  address: {
    city: "Delhi",
    pincode: 110001
  }
};

console.log(user.address.city); // Delhi
```

✂️ Object Destructuring

You can extract values directly:

```js
let { name, age } = student;
```

For nested objects:

```js
let {
  address: { city }
} = user;
```

🔁 Looping Through Objects

for-in loop

```js
for (let key in student) {
  console.log(key, student[key]);
}
```

Object.keys(), Object.values(), Object.entries()

```js
Object.keys(student);     // ["name", "age", "isEnrolled"]
Object.entries(student);  // [["name", "Ravi"], ["age", 21], ...]
```

📦 Copying Objects

Shallow Copy (one level deep)

```js
let newStudent = { ...student };
let newOne = Object.assign({}, student);
```

Deep Copy (nested levels)

```js
let deepCopy = JSON.parse(JSON.stringify(user));
```

> ❗ Note: JSON-based copy works only for plain data (no functions, undefined, etc.)

❓ Optional Chaining

Avoids errors if a nested property is undefined:

```js
console.log(user?.address?.city);    // Delhi
console.log(user?.profile?.email);  // undefined (no error)
```

🧠 Computed Properties

You can use variables as keys:

```js
let key = "marks";
let report = {
  [key]: 89
};
```

⚠️ Common Confusions

- Shallow copy copies only the first level
- `for-in` includes inherited keys (be cautious!)
- `delete obj.key` removes the property
- Spread ≠ deep copy

🧠 Mindset

Objects are **structured state** – perfect for modeling anything complex: a user, a form, a product, etc.
Use destructuring, chaining, and dynamic keys wisely.

🧪 Practice Zone

1. Create an object for a book (title, author, price)
2. Access properties using both dot and bracket
3. Write a nested object (user with address and location)
4. Destructure name and age from a student object
5. Loop through keys and values of an object
6. Convert object to array using `Object.entries()`
7. Copy an object using spread operator
8. Create a deep copy of an object with nested structure
9. Use optional chaining to safely access deep values
10. Use a variable as a key using computed properties