JavaScript learning path

Got it ✅
We'll restart the **programming part** from the absolute **basics of JavaScript** (how variables work) — but this time I'll make it **highly visual + structured**, so you can "see" how the engine runs things step by step.

🟢 Step 1: Variables in JavaScript (The Real Basics)

🧠 First Question: *What is a variable?*

- A **variable** is just a **named storage location in memory**.
- Think of it like a **box with a label** → the box stores some data, the label is the variable name.

🔎 How JS Creates Variables (Internally)

When JS code runs, it doesn't just go line by line immediately.
The **JS Engine** (e.g., V8 in Chrome/Node) does things in **two steps**:

1️⃣ Memory Creation Phase (Hoisting phase)

JS scans your code first and allocates memory for all variables and functions.

- `var` → memory is reserved, initialized with `undefined`.
- `let` / `const` → memory is reserved, but kept in a "**Temporal Dead Zone (TDZ)**" (they exist but can't be used yet).
- Functions → stored entirely in memory.

2️⃣ Execution Phase

JS runs line by line and assigns values to variables.

📊 Visual Representation

Take this code:

```js
console.log(a);
console.log(b);
console.log(c);

var a = 10;
let b = 20;
const c = 30;
```

🔹 Memory Creation Phase (before execution starts)

Variable	Where is it?	Value now
`a` (var)	In memory	`undefined`
`b` (let)	In memory	❌ In TDZ
`c` (const)	In memory	❌ In TDZ

(TDZ = Temporal Dead Zone, means "you can't touch me yet!")

🔹 Execution Phase (line by line)

1. `console.log(a);` → finds `a` → prints `undefined`.
2. `console.log(b);` → tries to find `b` → ReferenceError (TDZ).
3. `console.log(c);` → ❌ ReferenceError (TDZ).
4. `var a = 10;` → assigns `10` to `a`.
5. `let b = 20;` → moves `b` out of TDZ, assigns `20`.
6. `const c = 30;` → moves `c` out of TDZ, assigns `30`.

⚖️ Comparison of `var`, `let`, `const`

Feature	`var`	`let`	`const`
Scope	Function	Block	Block
Hoisting	✅ (undefined)	✅ (TDZ)	✅ (TDZ)

Feature	`var`	`let`	`const`
Re-declare	✅	❌	❌
Re-assign	✅	✅	❌
Best Use	❌ Avoid	✅ Temporary values	✅ Fixed references

💡 Developer Insight (Underrated but Important)

- `var` is dangerous → because it **ignores block scope** (like loops, ifs).

  ```js
  for (var i = 0; i < 3; i++) {}
  console.log(i); // 3 (oops, leaks outside loop)
  ```

- `let` respects block scope:

  ```js
  for (let j = 0; j < 3; j++) {}
  console.log(j); // ❌ ReferenceError
  ```

- `const` doesn't mean "frozen object" → it only freezes the **reference**, not contents:

  ```js
  const user = { name: "Alex" };
  user.name = "Sam";   // ✅ allowed
  user = { name: "John" }; // ❌ Error
  ```

🎯 Interview Tricky Question

```js
console.log(x);
var x = 5;
```

```
console.log(y);
let y = 10;
```

👉 **Answer:**

- `console.log(x)` → `undefined` (hoisted).
- `console.log(y)` → ❌ ReferenceError (TDZ).

✅ So far we've only touched **variables + memory model**.
Now, I want to ask you —
👉 Do you want me to **give you a few visual exercises on variables/hoisting** (so you can practice engine-thinking), or should we **move next into Data Types (primitive vs reference + hidden behaviors like stack vs heap, immutability, etc.)**?

Perfect 🔥
From now on, every topic we cover will have **practice exercises + interview-level tricky questions**. Let's continue with our current topic (**Variables, Hoisting, and TDZ**) and wrap it up with practice.

📝 Practice Exercises (Engine-Thinking)

👉 Try to **predict the output** before running in console.
I'll give **step-by-step hints** so you can trace the memory model.

Exercise 1: Hoisting Basics

```js
console.log(a);
var a = 100;

console.log(b);
let b = 200;
```

💡 **Hint:**

- During **memory creation phase**, `a` = `undefined`, `b` in TDZ.

- Execution runs top to bottom.

Exercise 2: Re-declaration

```js
var x = 10;
var x = 20;
console.log(x);

let y = 10;
let y = 20;
console.log(y);
```

💡 **Hint:**

- `var` allows re-declaration in same scope.
- `let` does not.

Exercise 3: Block Scope

```js
if (true) {
  var p = 1;
  let q = 2;
  const r = 3;
}
console.log(p);
console.log(q);
console.log(r);
```

💡 **Hint:**

- `var` ignores block scope.
- `let` and `const` are block-scoped.

Exercise 4: Const Objects

```js
```

```js
const obj = { lang: "JS" };
obj.lang = "Python";
console.log(obj);

obj = { lang: "C++" };
console.log(obj);
```

💡 **Hint:**

- `const` prevents **re-assignment**, not **mutation of properties**.

🎯 Interview-Level Tricky Questions

Q1. Temporal Dead Zone (TDZ) Trap

```js
let x = 10;
{
  console.log(x);
  let x = 20;
}
```

❓ What happens here?

Q2. Function Hoisting

```js
sayHello();

function sayHello() {
  console.log("Hello!");
}

sayHi();

var sayHi = function() {
  console.log("Hi!");
};
```

❓ Predict the output.

Q3. Weird `var` Behavior

```js
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

❓ Why does it print `3,3,3` instead of `0,1,2`?

✅ These exercises will **lock the idea of memory creation + execution + TDZ** into your brain.

👉 Do you want me to **first solve them with you (step-by-step engine explanation)** or should I let you **try them yourself and then we check answers together**?

Alright 🚀 let's go through **all exercises + tricky interview questions** one by one with **step-by-step engine simulation** (memory creation + execution).
This way you'll see **exactly why the output happens**.

📝 Exercise Solutions

✅ Exercise 1: Hoisting Basics

```js
console.log(a);
var a = 100;
```

```
console.log(b);
let b = 200;
```

Step 1 – Memory Creation Phase

- `var a` → reserved in memory, initialized as `undefined`.
- `let b` → memory reserved, but kept in **Temporal Dead Zone (TDZ)**.

Step 2 – Execution Phase

- `console.log(a);` → prints `undefined`.
- `a = 100;` → assignment happens.
- `console.log(b);` → ❌ Error → `ReferenceError: Cannot access 'b' before initialization`.

👉 Output

```javascript
undefined
ReferenceError
```

✅ Exercise 2: Re-declaration

```js
var x = 10;
var x = 20;
console.log(x);

let y = 10;
let y = 20;
console.log(y);
```

Explanation

- `var` allows re-declaration in the same scope → last assignment wins (`20`).
- `let` does NOT allow re-declaration in the same scope → ❌ `SyntaxError`.

👉 Output

```vbnet
```

```
20
SyntaxError: Identifier 'y' has already been declared
```

✅ Exercise 3: Block Scope

```js
if (true) {
  var p = 1;
  let q = 2;
  const r = 3;
}
console.log(p);
console.log(q);
console.log(r);
```

Explanation

- `var p` → function-scoped/global → accessible outside the block → prints `1`.
- `let q` & `const r` → block-scoped → not accessible outside → ❌ ReferenceError.

👉 Output

```javascript
1
ReferenceError
ReferenceError
```

✅ Exercise 4: Const Objects

```js
const obj = { lang: "JS" };
obj.lang = "Python";
console.log(obj);
```

```
obj = { lang: "C++" };
console.log(obj);
```

Explanation

- `obj.lang = "Python";` ✅ allowed → object mutation is fine.
- `obj = { lang: "C++" };` ❌ not allowed → re-assignment of a `const` variable.

👉 Output

```pgsql
{ lang: "Python" }
TypeError: Assignment to constant variable.
```

🎯 Interview Tricky Questions

✅ Q1. Temporal Dead Zone (TDZ) Trap

```js
let x = 10;
{
  console.log(x);
  let x = 20;
}
```

Memory Creation Phase (inside block)

- A **new scope** is created for block `{ }`.
- Inner `let x` shadows outer `let x`.
- This inner `x` is in TDZ until initialized.

Execution

- `console.log(x);` tries to access inner `x` (not outer one).
- But inner `x` is in **TDZ** → ❌ `ReferenceError`.

👉 Output

```pgsql
```

```
ReferenceError: Cannot access 'x' before initialization
```

✅ Q2. Function Hoisting

```js
sayHello();

function sayHello() {
  console.log("Hello!");
}

sayHi();

var sayHi = function() {
  console.log("Hi!");
};
```

Memory Creation Phase

- `sayHello` → fully hoisted as a function.
- `sayHi` → created as `undefined` (because it's `var`).

Execution

1. `sayHello();` ✅ works → prints `"Hello!"`.
2. `sayHi();` ❌ fails → at this moment `sayHi` is `undefined`.

👉 Output

```vbnet
Hello!
TypeError: sayHi is not a function
```

✅ Q3. Weird `var` Behavior

```js
```

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

Explanation

- `var i` is **function/global scoped**, not block scoped.
- So only **one** `i` **exists**, shared across all loop iterations.
- After loop finishes, `i = 3`.
- When callbacks finally run (after 1000ms), they all print `3`.

👉 **Output**

```
3
3
3
```

⚡ **Fix (with `let`)**

```js
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

- Now each iteration has its own block-scoped `i`.
- Prints `0,1,2`.

🔥 Big Takeaways

- **Hoisting** = vars `undefined`, functions fully hoisted, let/const in **TDZ**.
- **Block scope** vs **function scope** is critical for interviews.
- `var` pitfalls (`3,3,3` loop issue) are a classic **interview trap**.
- **Const ≠ immutability** → only prevents re-assignment, not mutation.

👉 Do you want me to now **give you the next batch of programming fundamentals** (like **data types & operators**) with engine-level details, or should we **do another practice round** on variables/hoisting before moving forward?