Hoisting and TDZ plan

Perfect 👌 Day 2 is a very important step, because **hoisting** and **temporal dead zone (TDZ)** are among the most frequently asked interview topics. Let's break your plan into structured sections: **learn → implement → practice problems**.

🔹 Day 2 – Hoisting & Temporal Dead Zone (Detailed Plan)

1. Learn Concepts

1.1 Variable Hoisting

- `var` declarations are hoisted (moved to the top of their scope), initialized with `undefined`.
- `let` / `const` are hoisted too, but **not initialized** → this is where **TDZ** happens.
- Example:

```js
console.log(a); // undefined (hoisted + initialized)
var a = 10;

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

console.log(c); // ❌ ReferenceError (TDZ)
const c = 30;
```

1.2 Function Hoisting

- **Function Declarations** → fully hoisted (you can call them before definition).
- **Function Expressions & Arrow Functions** → behave like variables (they're hoisted but uninitialized → TDZ if `let/const`, undefined if `var`).
- Example:

```js
// Function declaration
foo(); // ✅ works
function foo() {
  console.log("Hello");
```

```
}

// Function expression
bar(); // ❌ TypeError: bar is not a function
var bar = function() {
  console.log("Hi");
};

// Arrow function
baz(); // ❌ TypeError
let baz = () => console.log("Arrow");
```

1.3 Temporal Dead Zone (TDZ)

- The region between **hoisting** and **actual declaration** where accessing the variable throws `ReferenceError`.
- Example:

```js
{
  // TDZ starts
  console.log(a); // ❌ ReferenceError
  let a = 5; // TDZ ends here
  console.log(a); // ✅ 5
}
```

---

2. Implement (Hands-on)

Do small tests mixing different types of declarations/functions.

Case 1: var vs let vs const

```js
console.log(x); // undefined
var x = 1;

console.log(y); // ❌ ReferenceError
let y = 2;
```

```
console.log(z); // ❌ ReferenceError
const z = 3;
```

Case 2: Function declaration vs expression vs arrow

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

sayHello(); // ❌ TypeError
var sayHello = function() { console.log("Hello"); };

sayHey(); // ❌ ReferenceError
let sayHey = () => console.log("Hey");
```

Case 3: TDZ inside blocks

```js
{
  console.log(a); // ❌ ReferenceError
  let a = 10;
}
```

Case 4: Mixed var + function

```js
foo(); // ✅ Works
function foo() { console.log("function"); }

console.log(foo); // function definition
var foo = 42;
console.log(foo); // 42
```

---

3. Practice Problems (5–6 puzzles)

Try solving these step by step.

Puzzle 1

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

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

Puzzle 2

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

bar();
var bar = function() {
  console.log("bar");
};
```

Puzzle 3

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

Puzzle 4

```js
var x = 1;
function test() {
  console.log(x);
  var x = 2;
}
test();
```

Puzzle 5

```js
function sayHello() {
  console.log("Hello");
}
var sayHello;

sayHello(); // ?
```

Puzzle 6

```js
{
  var a = 1;
  let b = 2;
  const c = 3;
}
console.log(a); // ?
console.log(b); // ?
console.log(c); // ?
```

👉 By end of Day 2, you should:

- Clearly explain difference between **var/let/const hoisting**.
- Distinguish **function declarations vs expressions vs arrows**.
- Be able to predict **TDZ errors** in interview puzzles.

Do you want me to **solve the puzzles for you step by step**, or should I keep them unsolved so you can attempt first?

Great catch ⚡ This one looks simple but has a tricky hoisting detail. Let's break it down carefully.

The code

```js
function sayHello() {
  console.log("Hello");
}
var sayHello;

sayHello();
```

Step 1: Hoisting

When JavaScript parses this:

1. **Function declarations** are hoisted **with their full body**.
2. **Variable declarations** (`var sayHello;`) are also hoisted, but only the **declaration**, not the assignment.

So after hoisting, the engine "sees" it like this:

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

var sayHello; // declaration only (ignored because function already declared)
```

Step 2: Execution phase

- The function declaration is created first → `sayHello` now points to the function.
- The `var sayHello;` is just a declaration. Since `sayHello` is **already defined as a function**, the `var` line doesn't override it or reset it to `undefined`.
  (If it had an **assignment** like `var sayHello = 42;`, that would overwrite the function.)

So effectively the code running is:

```js
function sayHello() {
  console.log("Hello");
```

```
}

sayHello(); // ✅ calls the function
```

Step 3: Output

```nginx
Hello
```

✅ **The catch:**

- When both a **function declaration** and a **var declaration** exist with the same name, the function wins.
- The `var` is hoisted too, but it doesn't nullify the function unless you explicitly assign a new value later.

👉 Quick test:

```js
function sayHello() {
  console.log("Hello");
}
var sayHello = 42;

console.log(sayHello); // 42
sayHello(); // ❌ TypeError: sayHello is not a function
```

Do you want me to also compare this with the case when you use `let sayHello;` **instead of** `var`? That's even more interesting because TDZ comes into play.