JavaScript learning path

Awesome—here are the **solutions (Q1–Q10)** with **engine-accurate reasoning** (memory phase → execution phase), plus brief interview notes.

✅ Q1 — Hoisting + Shadowing

```js
var a = 5;
function test() {
  console.log(a);
  var a = 10;
  console.log(a);
}
test();
```

Memory phase (inside `test`): `var a` is hoisted → `a = undefined` (shadows outer `a`).
Execution:

- `console.log(a)` → `undefined`
- `a = 10`
- `console.log(a)` → `10`

Output

```javascript
undefined
10
```

Interview note: Inner `var` **shadows** outer binding and is hoisted as `undefined`.

✅ Q2 — `const` & Reassignment

```js
```

```
const arr = [1, 2, 3];
arr.push(4);
console.log(arr);

arr = [5, 6, 7];
console.log(arr);
```

- `arr.push(4)` mutates the array (allowed) → `[1,2,3,4]`
- `arr = [...]` tries to **reassign** the binding (illegal for `const`) → **TypeError**

Output

```
vbnet

[1, 2, 3, 4]
TypeError: Assignment to constant variable.
```

Interview note: `const` freezes the **binding**, not the **value**.

✅ Q3 — TDZ Trick

```
js

{
  console.log(x);
  let x = 100;
}
```

`let x` exists in the block but is in **TDZ** until its declaration line executes.

Output

```
pgsql

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

✅ Q4 — Global vs Block Scope

```
js
```

```
var x = 1;
let y = 2;

{
  var x = 10;
  let y = 20;
  console.log(x, y);
}

console.log(x, y);
```

- Inside block: `x` refers to the **same** `var x` (function/global-scoped) → becomes `10`; `y` is a new block `let` → `20`.
- Outside block: `x` is now `10`; outer `y` unchanged → `2`.

Output

```
10 20
10 2
```

Interview note: `var` ignores blocks; `let` is block-scoped.

✅ Q5 — Function Hoisting with `var` (Expression)

```
js

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

Memory phase: `sayHi` (var) → `undefined`. (Function **expression** not hoisted as a function.)
Execution: `sayHi()` while value is `undefined` → **TypeError** ("not a function").

Output

```
vbnet
```

```
TypeError: sayHi is not a function
```

✅ Q6 — Loop Trap (the classic)

```js
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 100);
}
```

Single `var i` shared; after loop ends `i = 3`. Timers fire later → log `3` three times.

Output

```
3
3
3
```

Fix: use `let i` (new binding per iteration) → prints `0,1,2`.

🧠 Q7 — Function vs `var` Hoisting Clash

```js
console.log(foo);
function foo() { return "I am a function"; }
var foo = "I am a variable";
console.log(foo);
```

Memory phase:

- Function declaration hoisted **as a function**.
- `var foo` hoisted as **existing** binding (already created), initialization to string happens at **runtime**.

Execution:

- First `console.log(foo)` → logs the **function**.

- Then `foo = "I am a variable"`.
- Second `console.log(foo)` → `"I am a variable"`.

Output

```javascript
ƒ foo() { ... }    // conceptually: the function object
I am a variable
```

Interview note: Function declarations win over `var` during creation; runtime assignment can overwrite them.

🧠 Q8 — Closure + `var/let` (timers & order)

```js
function counter() {
  for (var i = 0; i < 3; i++) {
    setTimeout(() => console.log("var:", i), i * 100);
  }
  for (let j = 0; j < 3; j++) {
    setTimeout(() => console.log("let:", j), j * 100);
  }
}
counter();
```

- `var` loop: one shared `i` → ends as `3` → prints `var: 3` three times.
- `let` loop: new `j` per iteration → prints `let: 0, 1, 2`.

Scheduling order (first loop schedules before second):

- ~0ms: `var: 3`, then `let: 0`
- ~100ms: `var: 3`, then `let: 1`
- ~200ms: `var: 3`, then `let: 2`

Output (typical)

```csharp
var: 3
let: 0
var: 3
```

```
let: 1
var: 3
let: 2
```

Interview note: Same-delay timers usually run in the order scheduled.

🧠 Q9 — TDZ with Default Parameters

```js
function test(x = y, y = 2) {
  console.log(x, y);
}
test();
```

Default params evaluate **left-to-right** in the **parameter scope**. When evaluating `x = y`, `y` isn't initialized yet → **TDZ**.

Output

```pgsql
ReferenceError: Cannot access 'y' before initialization
```

Interview note: Parameter default evaluation order matters.

🧠 Q10 — `const` object + `Object.freeze`

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

Object.freeze(obj);
obj.name = "ECMAScript";
console.log(obj);
```

- Mutating before freeze: allowed → `{ name: "JavaScript" }`.
- After `freeze`: properties become **non-writable, non-configurable**.

- In **non-strict mode**: assignment fails **silently**.
- In **strict mode**: throws **TypeError**.

Output (non-strict)

```
css

{ name: "JavaScript" }
{ name: "JavaScript" }
```

Interview note: `freeze` is **shallow**; nested objects remain mutable unless deep-frozen.

🔚 Rapid Recap

- Hoisting: `var → undefined`, `function → function`, `let/const → TDZ`.
- `var` shadows + leaks across blocks; `let/const` don't.
- Default parameters evaluate left-to-right (watch TDZ).
- `const` locks the binding; `freeze` locks (shallow) the object.
- Closures + `var` in loops cause the "3,3,3" bug.

Want another practice round that mixes **operators, coercion quirks (`==` vs `===`), and data types** next? That'll set us up perfectly for the **Data Types** lesson right after.