JavaScript variable scoping

```
function foo() {
  var a = b = 0;
  a++;
  return a;
}

console.log(foo());

console.log("type of a:", typeof a);
console.log("type of b:", typeof b);
```

Let’s carefully go step by step — this is a **very common tricky JavaScript interview question** 👇

🧩 The code

```js
function foo() {
  var a = b = 0;
  a++;
  return a;
}

console.log(foo());

console.log("type of a:", typeof a);
console.log("type of b:", typeof b);
```

🔍 Step-by-step explanation

Step 1: Line inside `foo()`

```js
var a = b = 0;
```

At first glance, it *looks like* both `a` and `b` are declared using `var`,
but **that’s not true** because of how JavaScript parses assignments.

This line is interpreted as:

```js
b = 0;
var a = b;
```

So here's what happens:

1. `b = 0` → assigns `0` to a variable `b`.
   But since there's **no `var`, `let`, or `const`** before `b`,
   it becomes a **global variable** (i.e. `window.b` in browsers or `global.b` in Node).
2. `var a = b;` → declares a **local variable** `a` inside `foo()` and sets it to the same value (`0`).

So after that line:

- `a` → **local** (inside `foo`) = `0`
- `b` → **global** = `0`

Step 2: `a++`

This increments the local `a` by `1`.
So `a` becomes `1`.

Then the function returns `a`.

Step 3: `console.log(foo());`

This prints the returned value, which is `1`.

✅ Output so far:

```
1
```

Step 4: After `foo()` executes

When the function finishes:

- `a` (local) is **destroyed** (not accessible outside `foo()`).
- `b` remains as a **global variable** (still exists outside).

Step 5: `console.log("type of a:", typeof a);`

Since `a` was declared with `var` **inside** `foo()`, it doesn't exist outside.

So `typeof a` → `"undefined"`

Step 6: `console.log("type of b:", typeof b);`

`b` is a **global variable** (implicitly created), so it **does exist** outside.

`typeof b` → `"number"`

✅ Final Output

```typescript
1
type of a: undefined
type of b: number
```

⚠️ Important Takeaway

If you want to avoid such confusion, **always declare variables explicitly**:

```js
function foo() {
  var a, b;
  a = b = 0; // both declared locally
  a++;
  return a;
}
```

Now both **`a`** and **`b`** are **local**, and outside **`foo()`**:

```js
typeof a // "undefined"
typeof b // "undefined"
```

Would you like me to visually show (via a small diagram) how the variables are stored in memory (global vs function scope)?

and in strict mode

Excellent question 👏 — this is exactly where **strict mode (`"use strict"`)** shows its power.

Let's analyze the same code **with strict mode enabled**.

🧩 Code (Strict Mode)

```js
"use strict";

function foo() {
  var a = b = 0;
  a++;
  return a;
}

console.log(foo());
console.log("type of a:", typeof a);
console.log("type of b:", typeof b);
```

🔍 Step-by-step Breakdown (with `"use strict"`)

Step 1: What `"use strict"` does

Strict mode makes JavaScript **less forgiving** — it prevents:

- Creation of **implicit globals**
- Use of **undeclared variables**
- Some silent errors

Step 2: Inside `foo()`

```
js

var a = b = 0;
```

Without strict mode, we saw that this becomes:

```
js

b = 0;   // creates a global variable (BAD!)
var a = b;
```

But in **strict mode**, this line now throws an **error** because:

> You're assigning to `b` without declaring it first (`var`, `let`, or `const`).

JavaScript will **not allow** you to implicitly create a global variable.

🚨 The Error

You'll get:

```
vbnet

ReferenceError: b is not defined
```

Execution stops right there — nothing below runs.

So no logs will print.

✅ Correct Way (in strict mode)

If you want this to work in strict mode, you must declare both variables:

```js
"use strict";

function foo() {
  var a, b;
  a = b = 0;
  a++;
  return a;
}

console.log(foo());
console.log("type of a:", typeof a);
console.log("type of b:", typeof b);
```

Now:

- Both **`a`** and **`b`** are **local**.
- After the function call:
  - **`typeof a`** → **`"undefined"`**
  - **`typeof b`** → **`"undefined"`**

✅ Output:

```typescript
1
type of a: undefined
type of b: undefined
```

🧠 Summary

Mode	Behavior	Output / Error
Non-strict	**b** becomes global implicitly	**1**, **`undefined`**, **`number`**
Strict mode	Implicit global creation not allowed	❌ **`ReferenceError: b is not defined`**

Would you like me to show a short visual comparison (like a table of variable scopes) between strict and non-strict versions?