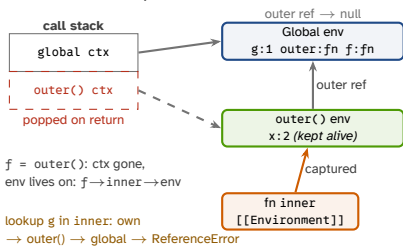
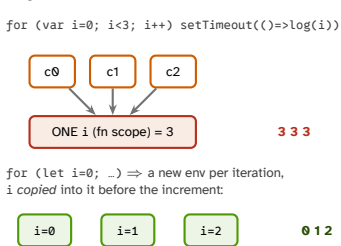


In one line: Scope is **lexical** — fixed by where code is *written*: every function keeps a hidden `[[Environment]]` pointer to the environment it was created in, and a closure is that pointer keeping **variables** (not values) alive. this is the opposite — **dynamic**, decided by *how the function is called* (except arrows).

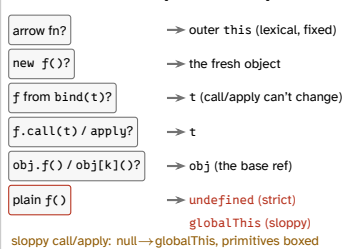
1 environments, not frames



2 for + setTimeout



3 what is this? (first match)



How it works — contexts and hoisting

- Each call pushes an **execution context**: a *LexicalEnvironment* (let/const/ class, blocks), a *VariableEnvironment* (var, function decls), the this binding, the realm. An **Environment Record** maps names → bindings and has an `[[OuterEnv]]` link: the **scope chain** (drawing 1).
- Hoisting** = bindings are created when the scope is entered, before any line runs (*FunctionDeclarationInstantiation*). What differs is the initial state:

declaration	at scope entry	use before the line	scope
var x	undefined	undefined	function
function f(){}	the whole function	works	function
let / const	uninitialised	ReferenceError (TDZ)	block
class C	uninitialised	ReferenceError (TDZ)	block
var f = ()=>{} f = undefined	f = undefined	TypeError: not a fn	function

* block-scoped in strict mode; sloppy block functions follow Annex B quirks.

- TDZ** runs from scope entry to the declaration *executing* — it is temporal, not textual: a function *called* after `let x=1` ran may read `x` even if written above it. `typeof x` in the TDZ throws too.
- Top-level `var/function` in a *script* become `globalThis` properties; `let/const/class` do not. Modules have their own scope.

Example — trace these

```
var a = 1;
function f() { log(a); var a = 2; } f(); // undefined
let b = 1;
{ log(b); let b = 2; } // ReferenceError: TDZ
log(typeof h); var h = 1; function h() {} // "function"
const o = { n: 'o', m() { return this.n; },
  later() { setTimeout(() => log(this.n)); } };
const m = o.m; m(); // strict: TypeError (this undefined)
(o.m)(); // 'o' -- parens keep the reference
(0, o.m)(); // loses this: comma yields a value
o.later(); // 'o' -- arrow reads later()'s this
o.m.bind({n:'x'}).call(o); // 'x' -- bind is permanent
function F() { log(this.n); }
new (F.bind({n:'x'}))(); // undefined -- new beats bind
```

Closures — what is captured

- The **binding**, by reference: later writes are visible to every closure sharing it. To freeze a value, pass it as an argument or copy it into a block const. (React's stale closure is this — see react-hooks-deep.)
- Swift**: also captures variables by reference; a *capture list* `[x]` copies at creation — JS has none. No `[weak self]` either: a tracing GC collects cycles; leaks are *reachability* (listener/timer/cache holds the closure).
- V8: closures of one scope share one **Context** object with the captured vars — a big array used by *any* inner fn lives while *any*

of them does.

this — the rules (drawing 3)

- Arrows** have no own this, arguments, super, new.target, prototype; call/apply/bind ignore their `thisArg`; new throws. In an object *literal* an arrow sees the *outer* this.
- Lost this**: `arr.forEach(o.m)`, `const {m} = o`, `setTimeout(o.m)` pass the *function*, not the reference. Fix: `o.m.bind(o)`, a wrapper arrow, or an arrow **class field** `onTap = () => this.go()` — bound for life, but one fn *per instance*, not on the prototype (no `super.onTap`, no prototype spy).
- Top level: this is undefined in an ES module, `module.exports` in CommonJS, `globalThis` in a sloppy script.

Strict mode ('use strict'; classes + modules always)

Plain-call this = undefined · assigning an undeclared name throws (sloppy makes a global) · writing a read-only property throws (sloppy ignores) · no with, duplicate params, 010 · arguments stops aliasing params · eval gets its own scope.

Interview traps

- ! “let/const aren’t hoisted” — they are, *uninitialised* (TDZ).
- ! “A closure copies the value” — it keeps the variable (3 3 3); let fixes it with a fresh binding per iteration.
- ! this is the *call site*, not where the fn was defined — only arrows and bind fix it. bind twice: the first wins; new overrides it.

Remember

Scope is where it’s written · this is how it’s called · closures keep variables, not values · arrows borrow this.

Likely questions

- Closure? — a function + the environment it was created in, kept alive by the function.
- Why 3 3 3? — one function-scoped var i; callbacks run after the loop ends.
- this priority? — new > bind > call/apply > method call > default.
- Class method’s this undefined in a callback? — detached, and class code is strict.