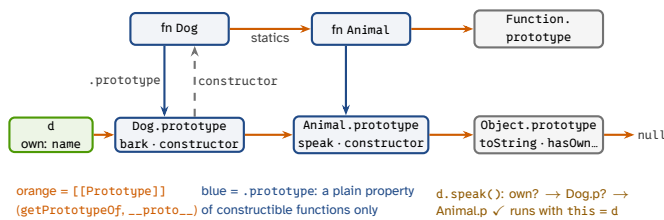
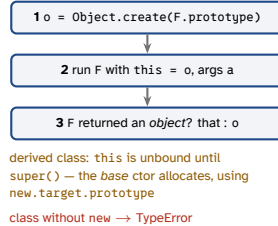


In one line: Every object has a hidden `[[Prototype]]` link; a read that misses *own* properties walks the chain to null — objects **delegate** at run time, nothing is copied. `class` is mostly syntax over constructor functions and their `.prototype`.

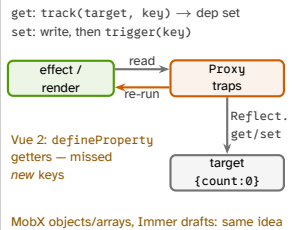
1 `class Dog extends Animal; d = new Dog()`



2 `new F(a)` step by step



3 Proxy reactivity (Vue 3)



How it works

- Read** `o.k`: `own` property? else `[[Prototype]]`'s, ... else undefined. A getter found up the chain runs with `this = o` (the receiver).
- Write** `o.k = v` creates/updates an *own* property — **shadowing** the inherited one — *unless* the chain has a **setter** for `k` (it runs) or a **non-writable** `k` (write fails; throws in strict).
- `Object.create(proto)` links without a constructor; `Object.create(null)` = a dictionary with no inherited keys. `setPrototypeOf` later deoptimises.
- `Own`: `Object.hasOwn`, `Object.keys` (own enumerable strings) · chain: `k in o`, `for...in`. `instanceof C` = `C` is `C.prototype` on the chain — fails across realms (use `Array.isArray`).
- Swift**: a protocol is checked at compile time; a prototype is a live object — patch `Array.prototype` and every array changes now.

Classes — sugar, except...

```
class Counter extends Base {
  #n = 0; // private: a brand, not a property
  static { /* static init block: runs once */ }
  inc = () => ++this.#n; // own field, one fn per instance
  get value() { return this.#n; } // accessor on the prototype
  static is(o) { return #n in o; } // brand check
}
```

- Not sugar**: no call without `new`; strict body; methods non-enumerable and not constructible; TDZ; `super()` before `this`; `extends` `Array/Error` makes real exotic instances (ES5 couldn't).
- #private**: invisible to reflection; `this.#n` through a Proxy of the instance throws `TypeError` (not forwarded).
- super** comes from the method's `[[HomeObject]]` (fixed at definition), not `this`: a copied method keeps it.

Descriptors and locking

Data {value, writable, enumerable, configurable} · accessor {get, set, enumerable, configurable}.
Literal/assignment: all true; `defineProperty`: omitted = false.
configurable: false: no delete, no redefine (writable may still go true → false).

	add	delete	change value	sets on every own prop
<code>preventExtensions</code>	x	✓	✓	—
<code>seal</code>	x	x	✓	<code>configurable: false</code>
<code>freeze</code>	x	x	x	<code>+writable: false</code>

All **shallow** (nested objects stay mutable); setters still run; violations are silent in sloppy, throw in strict.

Proxy + Reflect

```
const logged = t => new Proxy(t, {
  get: (t, k, r) => (log('get', k), Reflect.get(t, k, r)),
  set: (t, k, v, r) => Reflect.set(t, k, v, r) }); // false: throws
```

`r` = receiver: getters see the proxy as this. 13 traps (has `deleteProperty` `ownKeys` apply `construct...`); `Reflect.x` = trap `x`'s default.

Symbols, iterators, generators

- `Symbol()` = unique key, skipped by `Object.keys/for...in/JSON`. Well-known hooks: `Symbol.iterator` (for-of, spread, destructuring), `asyncIterator`, `toPrimitive`.
- Iterator**: `next()` → {value, done}; lazy; optional `return()` runs on break (cleanup).
- function***: calling it runs *nothing*; each `next(v)` resumes to the next `yield`, and `v` becomes the *value of the paused yield* — a coroutine (redux-saga; co, pre-async/await).

```
function* g() { const a = yield 1;
  const b = yield a + 1; return a + b; }
const it = g(); // nothing ran yet
it.next('x'); // {value: 1, done: false} first arg is ignored
it.next(10); // a = 10 -> {value: 11, done: false}
it.next(5); // b = 5 -> {value: 15, done: true}
it.next(); // {value: undefined, done: true}
```

Mixins vs inheritance

One `extends`; a mixin is a class factory `B => class extends B {...}` — `extends Ser(Stamped(Model))` keeps `super`. Swift: protocol extensions.

Interview traps

- ! Instances link to `Dog.prototype`; `Dog` itself links to `Animal`.
- ! `P.prototype.tags = []`: one array shared by every instance.
- ! Spread/assign copy own enumerable props, run getters, drop the prototype.

Remember

Own first, then up the chain · writes shadow · .prototype is for instances · freeze is shallow · Reflect is the default trap.

Likely questions

- Why pass receiver? — inherited getters must see the proxy as this.
- Generator vs `async`? — `async` = generator yielding promises + auto-runner.