



MÓDULO DWEC · DAW — UNIDAD DIDÁCTICA 4

Objetos y Programación Orientada a Objetos

RA3 · Objetos del lenguaje, prototipos, clases, herencia y encapsulación

- Entender qué hay debajo de class: objetos y prototipos
- Modelar con clases, herencia y campos privados con criterio
- Elegir entre herencia y composición, y usar bien los objetos nativos

pedido (instancia)

id: 1042

total: 59.90

[[Prototype]]

Pedido.prototype

calcularTotal: *f*

añadir: *f*

[[Prototype]]

Object.prototype

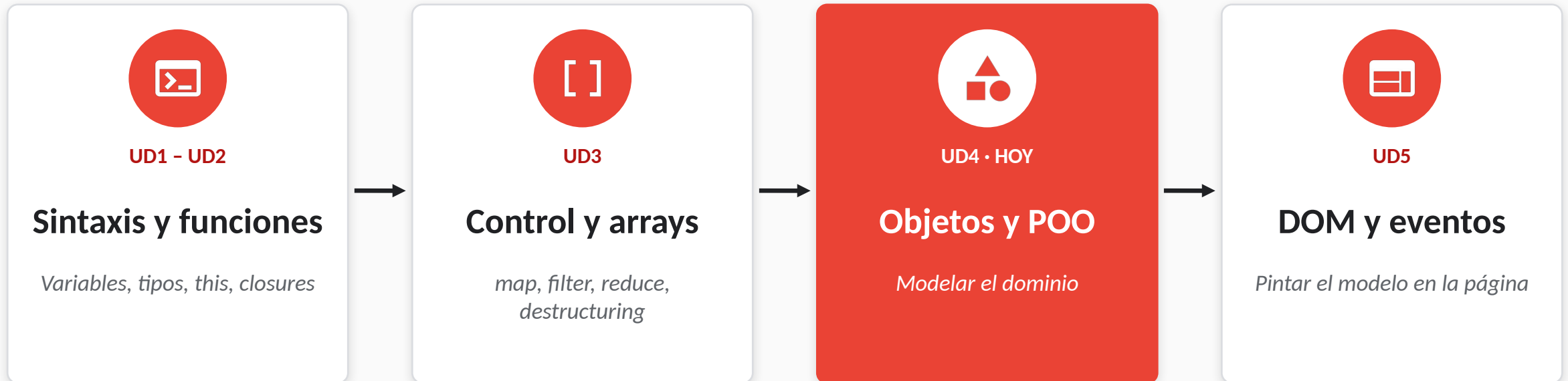
toString: *f*

hasOwnProperty: *f*



DÓNDE ESTAMOS

De datos sueltos a modelos con comportamiento



La pregunta de la unidad: ¿cómo represento en código las cosas de mi aplicación (usuarios, productos, pedidos) con sus datos y lo que saben hacer?

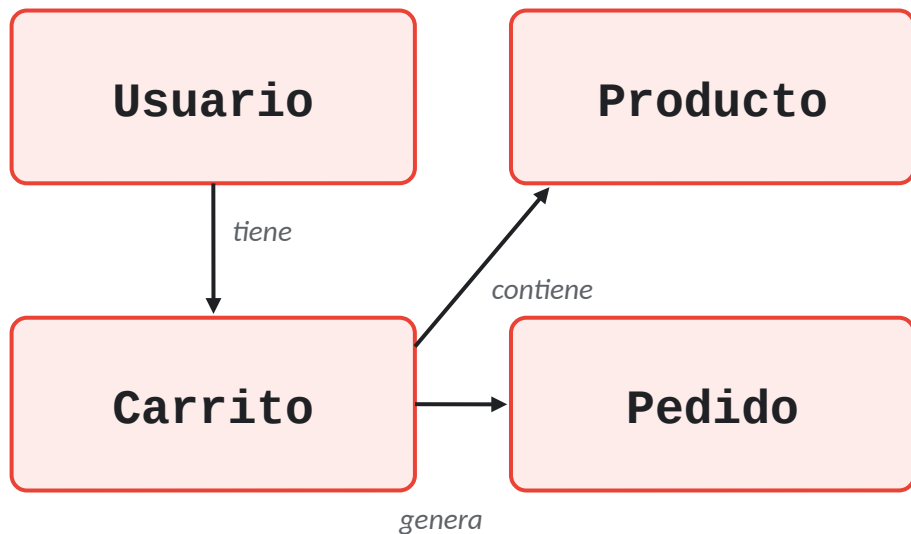


OBJETIVO — La Práctica 2 (06/11) os pide un modelo con clases propias que luego pintaréis con el DOM (UD5). Hoy se construye la mitad.



¿Por qué esto importa en el mundo real?

El dominio de una tienda online



Cada caja = datos + lo que sabe hacer

`carrito.añadir(producto) · pedido.calcularTotal()`



Modelar el dominio

Usuarios, productos, pedidos: la POO da forma a los conceptos del negocio



Lo vais a leer a diario

Error, Date, Map, HTMLElement... las APIs del navegador son clases



JavaScript es distinto

Por debajo usa prototipos, no clases como Java o C#



Un mal diseño cuesta caro

Jerarquías rígidas = código difícil de cambiar después



Debate vivo

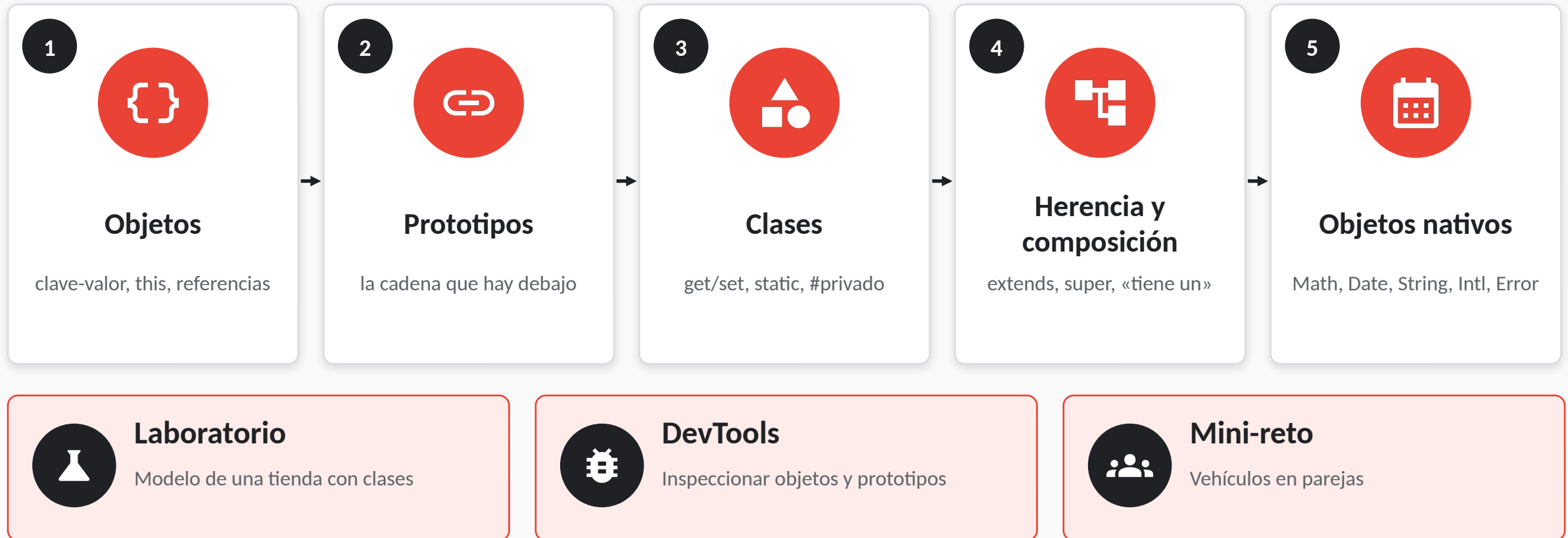
«Composición frente a herencia» se discute en cada proyecto



DATO CLAVE — «Favorece la composición sobre la herencia» (Gamma et al., Design Patterns, 1994) es uno de los principios de diseño más citados.



Cinco bloques, de abajo arriba



 **CLAVE** — class no es magia: es una forma cómoda de escribir objetos y prototipos. Por eso empezamos por abajo.



BLOQUE 1

Objetos: datos con nombre

Clave-valor, acceso, métodos y this, referencias, recorrer y proteger



Un objeto es un conjunto de pares clave → valor

usuario.js

```
const usuario = {  
  nombre: "Ana",  
  edad: 31,  
  activo: true,  
  roles: ["editor", "admin"],  
  direccion: { ciudad: "Sevilla" },  
  saludar() {  
    return `Hola, soy ${this.nombre}`;  
  },  
};
```

CLAVE	VALOR	TIPO
nombre	"Ana"	string
edad	31	number
activo	true	boolean
roles	[...]	array (objeto)
direccion	{...}	otro objeto
saludar	f	función → método

Propiedad = clave + valor. **Método** = propiedad cuyo valor es una función. Las claves son siempre strings (o Symbol).

NOTA — Un array, una fecha o una función también son objetos. En JS, todo lo que no es primitivo es un objeto.



Punto o corchetes: leer, añadir, cambiar y borrar

Notación punto

```
usuario.nombre
```

- La clave es un identificador válido
- Es la habitual: legible y corta

Notación corchetes

```
usuario[clave]
```

- La clave está en una variable
- Tiene guiones, espacios o empieza por número
- Se calcula en tiempo de ejecución

Ya lo conocéis de la UD3: `usuario?.direccion?.ciudad` no revienta si falta `direccion`, y `const { nombre, edad } = usuario` saca varias propiedades a la vez.

```
acceso.js

const campo = "email";
usuario[campo];           // como usuario.email
usuario["fecha-alta"];    // con guion: corchetes

usuario.telefono = "600..."; // añade
usuario.edad = 32;           // cambia
delete usuario.activo;      // borra
usuario.dni;                // undefined

// atajos al crear (ES2015)
const nombre = "Luis";
const p = { nombre, [campo]: "l@x.es" };
// { nombre: "Luis", email: "l@x.es" }
```



CLAVE — Leer una propiedad que no existe NO da error: devuelve `undefined`. El error llega al hacer `undefined.algo`.



this = el objeto que está a la izquierda del punto

contador.js

```
const contador = {  
  valor: 0,  
  incrementar() {  
    this.valor++;  
    return this.valor;  
  },  
};
```

```
contador.incrementar(); // 1 → this = contador
```

¿Quién es this en la llamada?

contador

incrementar()

this

Se decide en el momento de la LLAMADA, no al escribir el método.



Trampa (UD2): una función flecha NO tiene this propio. incrementar: () => this.valor++ no funciona: this sería el de fuera (undefined en un módulo). Los métodos de objetos y clases se escriben con la sintaxis corta metodo() { ... }.

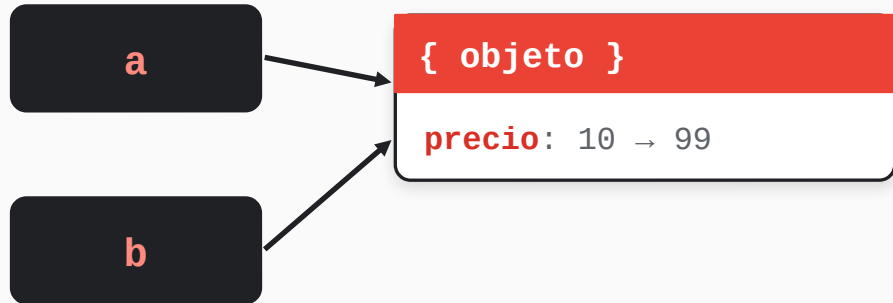


NOTA — Volveremos a this en el bloque 3: al pasar un método como callback se pierde. Pasa muchísimo con eventos (UD5).



La variable no guarda el objeto: guarda una flecha hacia él

```
const b = a;
```



b.precio = 99 → a.precio también es 99

```
const c = { ...a };
```



Copia SUPERFICIAL: los objetos de dentro siguen compartidos

Operación	¿Qué hace?	Úsala para...
b = a	Copia la flecha (mismo objeto)	Compartir a propósito
{ ...a }	Objeto nuevo, 1 nivel de copia	Cambiar propiedades sin tocar el original
structuredClone(a)	Copia profunda (todos los niveles)	Clonar datos anidados (no funciones ni clases)
a === b	¿Es la MISMA flecha?	{ x: 1 } === { x: 1 } → false



CLAVE — Recordad la UD2: primitivos por valor, objetos por referencia. Pasar un objeto a una función permite que la función lo modifique.



Object.keys, values y entries: el objeto como lista



recorrer.js

```
for (const [prod, uds] of Object.entries(stock)) {  
  console.log(`${prod}: ${uds}`);  
}  
const agotados = Object.keys(stock).filter(k => stock[k] === 0);  
const conIva = Object.fromEntries(  
  Object.entries(precios).map(([k, v]) => [k, v * 1.21]));
```

¿Y for...in?

- También recorre las claves heredadas del prototipo (bloque 2)
- Por eso hoy se prefiere Object.keys / entries + for...of
- Nunca for...in sobre arrays (UD3)



CLAVE — Con entries + map + fromEntries transformáis un objeto con lo que ya sabéis de arrays (UD3).



const, Object.seal y Object.freeze no son lo mismo

	Reasignar la variable	Cambiar valores	Añadir propiedades	Borrar propiedades
<code>const obj</code>	X	✓	✓	✓
<code>Object.seal(obj)</code>	—	✓	X	X
<code>Object.freeze(obj)</code>	—	X	X	X

config.js

```
const CONFIG = Object.freeze({ iva: 0.21, moneda: "EUR" });
CONFIG.iva = 0;                // en un módulo: TypeError
CONFIG = {};                   // TypeError: assignment to constant

const datos = Object.freeze({ dir: { cp: "41001" } });
datos.dir.cp = "28001"; // ¡funciona! freeze es superficial
```

Lo que hay que llevarse

- const protege la VARIABLE, no el objeto
- freeze protege el objeto, pero solo el primer nivel
- Fuera de modo estricto el cambio se ignora en silencio; en módulos y clases lanza TypeError



NOTA — Uso típico de freeze: constantes de configuración y «enumerados» (`Object.freeze({ PENDIENTE: "pendiente", ENVIADO: "enviado" })`).

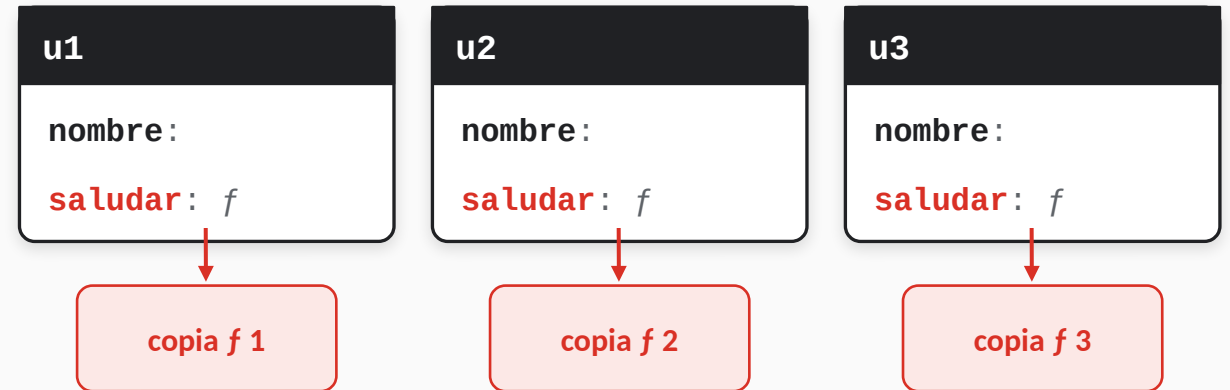


Una función fábrica... y un problema de memoria

```
fabrica.js

function crearUsuario(nombre) {
  return {
    nombre,
    saludar() {
      return `Hola, ${this.nombre}`;
    },
  };
}

const u1 = crearUsuario("Ana");
const u2 = crearUsuario("Luis");
```



Cada objeto lleva SU PROPIA copia de saludar: 1 000 usuarios = 1 000 funciones idénticas.



¿Y si todos los usuarios compartieran UN solo saludar? Eso es exactamente lo que hace JavaScript con los prototipos. Bloque 2.



NOTA — Las fábricas siguen siendo útiles (con closures dan privacidad, UD2). El problema es solo repetir métodos en muchos objetos.



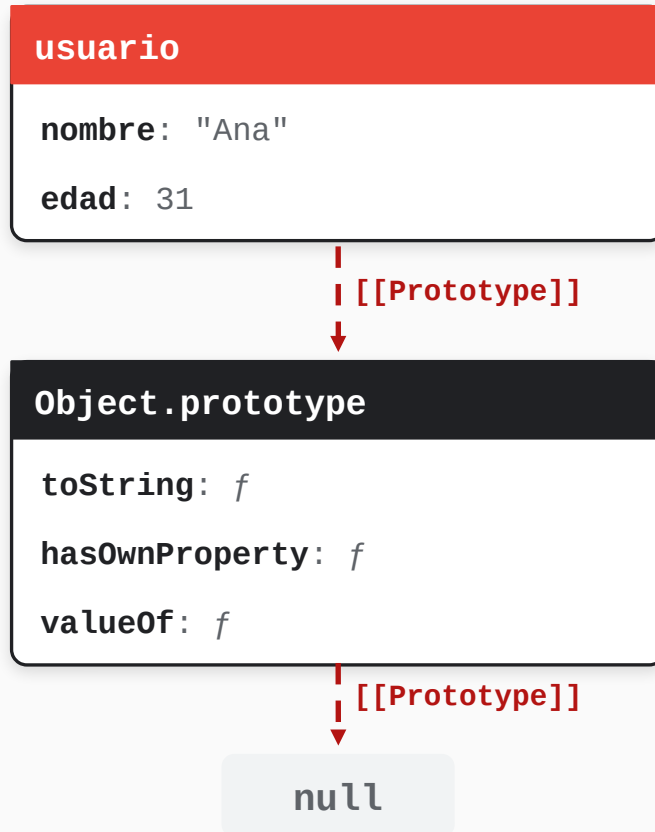
BLOQUE 2

Prototipos: lo que hay debajo

[[Prototype]], la cadena, new, métodos compartidos y por qué class es azúcar sintáctico



Todo objeto tiene un enlace oculto a otro objeto: su prototipo



```
consola

const usuario = { nombre: "Ana", edad: 31 };
usuario.toString(); // "[object Object]" ¿de dónde sale?
Object.getPrototypeOf(usuario) === Object.prototype; // true

const lista = [1, 2];
// lista → Array.prototype (map, filter...) → Object.prototype → null
```

[[Prototype]]: enlace interno; se lee con `Object.getPrototypeOf(obj)` (en consola aparece como `[[Prototype]]`).

__proto__: forma antigua de acceder a ese enlace. Funciona, pero está desaconsejada.

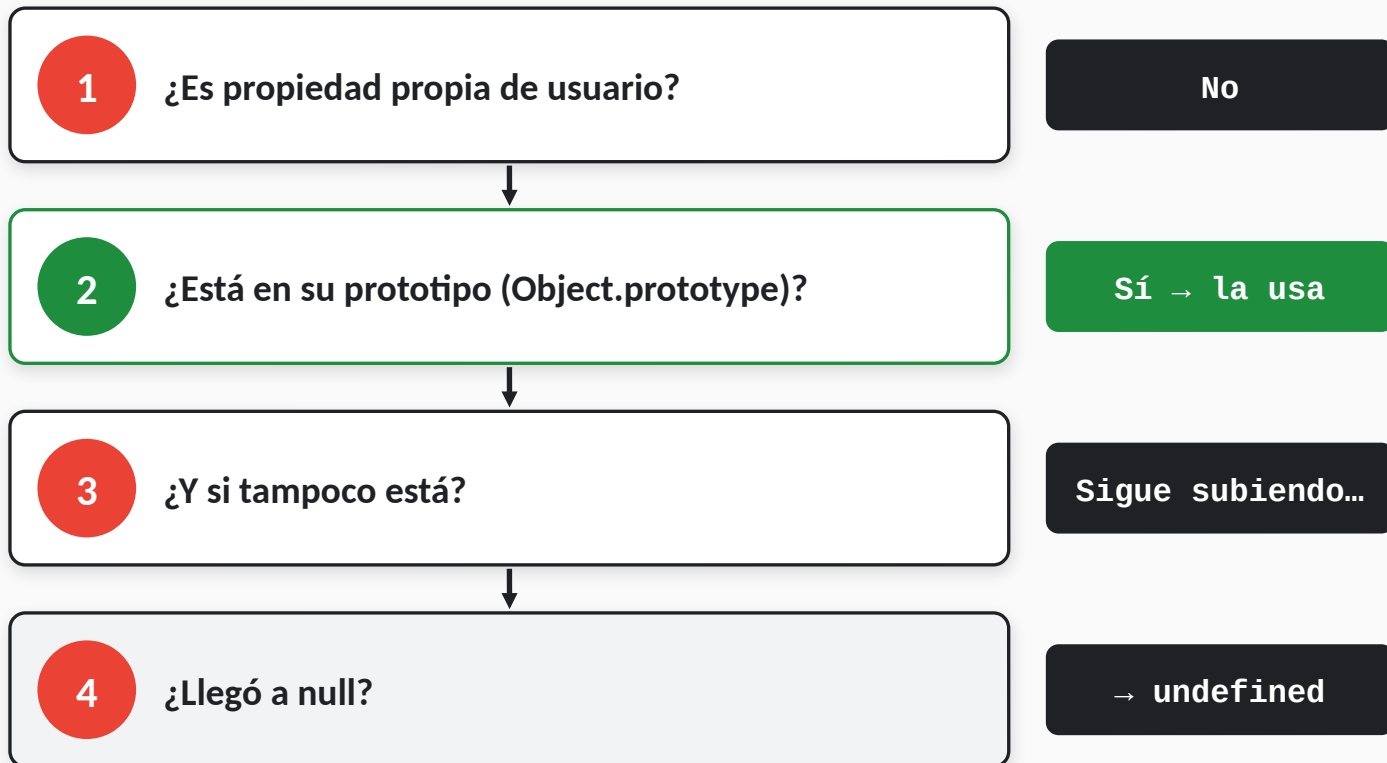
Cadena de prototipos: el enlace del enlace del enlace... hasta llegar a null.



CLAVE — Por eso `[1,2].map` existe sin que lo hayáis escrito: `map` vive en `Array.prototype` y todos los arrays apuntan a él.



¿Cómo encuentra JavaScript usuario.toString?



```
propias.js

"nombre" in usuario;           // true
"toString" in usuario;         // true
(heredada)
Object.hasOwn(usuario, "nombre"); // true
Object.hasOwn(usuario, "toString"); // false
```

Sombrear (shadowing)

Si el objeto tiene una propiedad propia con el mismo nombre, se usa esa y la búsqueda se para: así se «sobrescribe» un método heredado.



CLAVE — LEER sube por la cadena. ESCRIBIR (`obj.x = 1`) siempre crea o cambia la propiedad en el propio objeto, nunca en el prototipo.



Función constructora + new: la forma clásica (antes de 2015)

persona.js

```
function Persona(nombre, edad) {  
  this.nombre = nombre;  
  this.edad = edad;  
}  
Persona.prototype.saludar = function () {  
  return `Hola, soy ${this.nombre}`;  
};  
  
const p = new Persona("David", 30);  
p.saludar(); // "Hola, soy David"
```

Lo que hace new Persona("David", 30), paso a paso

1

Crea un objeto vacío

```
{ }
```

2

Lo enlaza al prototipo

```
[[Prototype]] →  
Persona.prototype
```

3

Ejecuta la función con this = ese objeto

```
this.nombre = "David"...
```

4

Lo devuelve

```
p
```



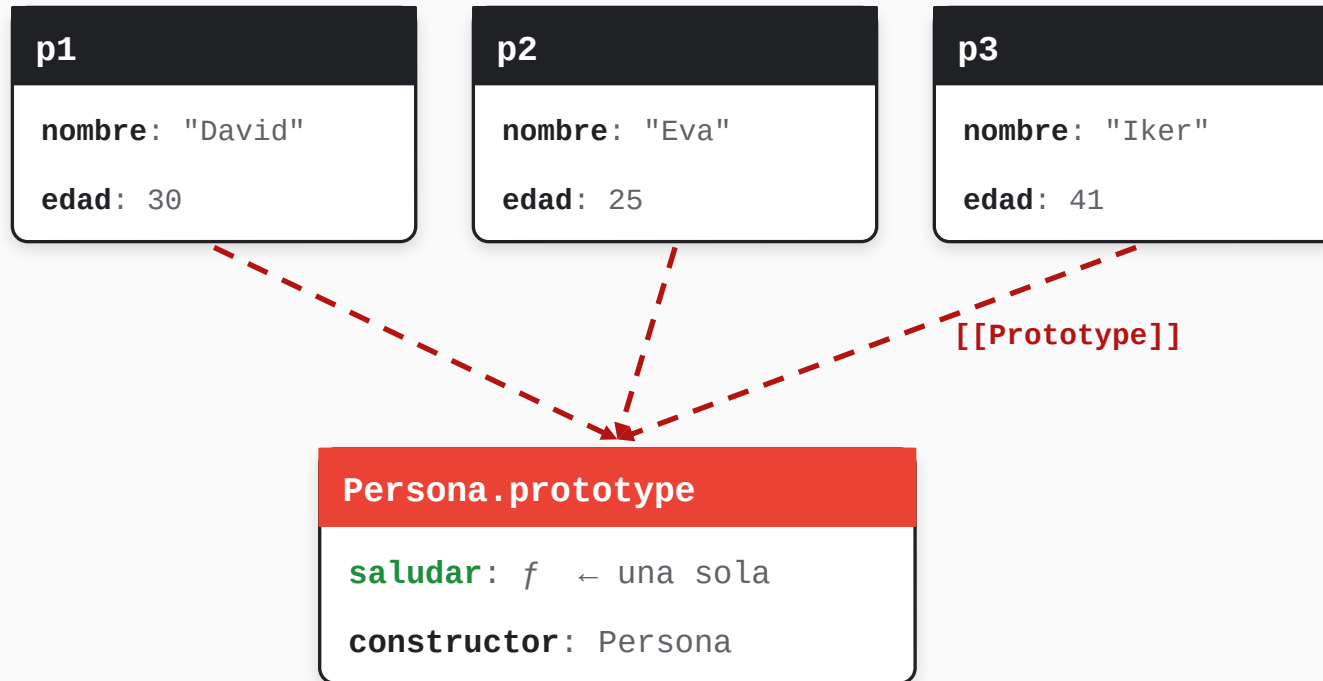
Sin new no funciona: Persona("David", 30) ejecuta la función con this = undefined (modo estricto) y lanza TypeError; en código antiguo no estricto ensuciaba el objeto global. **Convención: las constructoras y clases empiezan por Mayúscula.**



NOTA — No vais a escribir constructoras nuevas, pero las encontraréis en código heredado y en tutoriales antiguos. Hay que saber leerlas.



Los métodos viven UNA vez en el prototipo y todos los usan



Comparado con la fábrica

	Fábrica	Prototipo
Datos (nombre, edad)	en cada objeto	en cada objeto
Métodos	copiados en cada uno	1 vez, compartidos
1 000 objetos	1 000 funciones	1 función

Si cambias `Persona.prototype.saludar`, cambia para TODAS las instancias, también las ya creadas.

No toquéis los prototipos nativos (`Array.prototype...`): rompe librerías.



CLAVE — Regla: los DATOS van en cada objeto (`this.x` en el constructor); el COMPORTAMIENTO va en el prototipo.



Crear con un prototipo concreto e inspeccionar la cadena

object-create.js

```
const animal = {
  describir() {
    return `${this.nombre} hace ${this.sonido}`;
  },
};
const gato = Object.create(animal); // gato → animal
gato.nombre = "Misi";
gato.sonido = "miau";
gato.describir(); // "Misi hace miau"
```

gato

nombre: "Misi"
sonido: "miau"

↓
[[Prototype]]
↓

animal

describir: f

Herencia entre objetos, sin clases ni constructoras: así de simple es el modelo real.

Herramienta	Para qué
<code>Object.getPrototypeOf(obj)</code>	Ver el prototipo de un objeto
<code>Object.create(proto)</code>	Crear un objeto con ese prototipo
<code>obj instanceof Clase</code>	¿Está Clase.prototype en la cadena de obj?
<code>console.dir(obj)</code>	Ver el objeto desplegable con su [[Prototype]]



NOTA — `Object.setPrototypeOf` existe, pero cambiar el prototipo de un objeto ya creado es lento. Mejor fijarlo al crear.



class escribe lo mismo, de forma más clara y segura

antes.js (función constructora)

```
function Persona(nombre) {  
  this.nombre = nombre;  
}  
Persona.prototype.saludar = function () {  
  return `Hola, soy ${this.nombre}`;  
};
```

=

ahora.js (class, ES2015)

```
class Persona {  
  constructor(nombre) {  
    this.nombre = nombre;  
  }  
  saludar() {  
    return `Hola, soy ${this.nombre}`;  
  }  
}
```

```
typeof Persona; // "function" (en los dos casos)
```

```
Persona.prototype.saludar; // f (en los dos casos)
```

Lo que class añade de verdad: obliga a usar new, el cuerpo va siempre en modo estricto, los métodos no salen en for...in, y permite #privados, static y extends/super cómodos.



CLAVE — Azúcar sintáctico = otra forma de escribir algo que ya existía. Debajo de class sigue habiendo una función y un prototipo.



BLOQUE 3

Clases

Anatomía, constructor, getters y setters, campos privados, static y el this perdido



Las piezas de una clase

Producto.js

```
class Producto {  
  categoria = "general"; // ① campo público  
  
  constructor(nombre, precio) { // ② constructor  
    this.nombre = nombre;  
    this.precio = precio;  
  }  
  
  precioConIva() { // ③ método  
    return this.precio * 1.21;  
  }  
}  
  
const teclado = new Producto("Teclado", 50); // ④
```

Producto

+ categoria
+ nombre
+ precio

+ precioConIva()

- ① Campo: propiedad con valor inicial; se crea en cada instancia
- ② constructor: se ejecuta solo con new; recibe los datos
- ③ Método: va al prototipo (compartido)
- ④ Instancia: un objeto concreto creado con new



NOTA — En UML: nombre arriba, atributos en medio, métodos abajo. + público, - privado, subrayado o «static» para lo estático.



El constructor deja el objeto en un estado válido... o no lo crea

Producto.js

```
class Producto {  
  constructor(nombre, precio = 0, stock = 0) {  
    if (!nombre?.trim()) {  
      throw new Error("El nombre es obligatorio");  
    }  
    if (precio < 0) {  
      throw new RangeError("Precio negativo");  
    }  
    this.nombre = nombre.trim();  
    this.precio = precio;  
    this.stock = stock;  
  }  
}
```

```
new Producto("Ratón", 15)
```

✓ objeto válido

```
new Producto("", 15)
```

✗ Error: nombre obligatorio

```
Producto("Ratón", 15)
```

✗ TypeError: Class constructor Producto cannot be invoked without 'new'

Además: las clases NO se elevan (hoisting) como las funciones: usarlas antes de su línea da ReferenceError (zona muerta temporal, UD2). Una clase por fichero y se importa.



CLAVE — Valores por defecto en los parámetros y validación al principio: si algo no cuadra, se lanza un error y el objeto no llega a existir.



Parece una propiedad, pero por dentro es un método

Producto.js

```
class Producto {
  #precio = 0;
  constructor(nombre, precio) {
    this.nombre = nombre;
    this.precio = precio; // ← pasa por el set
  }
  get precio() {
    return this.#precio;
  }
  set precio(valor) {
    if (valor < 0) throw new RangeError("Precio negativo");
    this.#precio = valor;
  }
  get precioConIva() { // calculada, solo lectura
    return this.#precio * 1.21;
  }
}
```

p.precio

→ get precio()

lee #precio

p.precio = 20

→ set precio(20)

valida y guarda

p.precio = -5

→ set precio(-5)

RangeError

p.precioConIva = 9

→ no hay set

TypeError (modo
estricto)

Solo cuando aportan algo: validar, calcular o proteger. Un get/set que solo copia el valor sobra.



CLAVE — Se usa sin paréntesis: p.precioConIva, no p.precioConIva(). El get no debe tener efectos secundarios ni ser lento.



Campos y métodos privados con

CuentaBancaria.js

```
class CuentaBancaria {  
  #saldo = 0; // campo privado  
  constructor(titular) {  
    this.titular = titular; // público  
  }  
  ingresar(cantidad) {  
    this.#validar(cantidad);  
    this.#saldo += cantidad;  
  }  
  get saldo() { return this.#saldo; }  
  
  #validar(cantidad) { // método privado  
    if (cantidad <= 0) throw new RangeError("Cantidad");  
  }  
}
```

DENTRO de la clase

#saldo

#validar()

solo accesibles desde los métodos de la propia clase

ingresar()

get saldo

↑ interfaz pública: la única puerta

FUERA

```
cuenta.ingresar(100); cuenta.saldo; // ✓ 100
```

```
cuenta.#saldo = 1e6; // ✗ SyntaxError
```



CLAVE — Encapsular = esconder el estado y exponer solo métodos que lo cambian de forma controlada. Nadie puede dejar el saldo en negativo.



#privado es el único que el lenguaje hace cumplir en clases

	_saldo (convención)	closure (UD2)	#saldo (ES2022)
¿Accesible desde fuera?	Sí, cualquiera	No	No
¿Quién lo impide?	Nadie: es un aviso	El ámbito de la función	El propio lenguaje
Error al intentarlo	Ninguno	Es undefined	SyntaxError al cargar
¿Sale en Object.keys / JSON?	Sí	No	No
Dónde lo verás	Código antiguo	Fábricas, módulos	Clases modernas

Es **SyntaxError**, no un error normal

Se detecta al leer el fichero, antes de ejecutarlo: si escribís `obj.#saldo` fuera de la clase, NO se ejecuta nada de ese script. No se puede capturar con `try/catch`.

Ojo con la consola de Chrome

Para depurar, la consola de DevTools SÍ deja leer `cuenta.#saldo`. No significa que sea público: en vuestro código fuente fallará.



NOTA — Para comprobar si un objeto tiene un privado sin romper nada: `#saldo in obj` (dentro de la clase) devuelve `true/false`.



Lo estático pertenece a la clase, no a cada instancia

Pedido.js

```
class Pedido {  
  static #contador = 0;  
  static IVA = 0.21;  
  
  constructor(lineas) {  
    this.id = ++Pedido.#contador;  
    this.lineas = lineas;  
  }  
  
  static desdeJSON(texto) { // método fábrica  
    const datos = JSON.parse(texto);  
    return new Pedido(datos.lineas);  
  }  
}  
  
Pedido.IVA; // 0.21  
Pedido.desdeJSON(json).id; // 1
```

Pedido (la clase)

IVA

#contador

desdeJSON()

*una sola copia, compartida***pedido 1**

id: 1

lineas: [...]

pedido 2

id: 2

lineas: [...]

pedido1.IVA → undefined · Pedido.IVA → 0.21

Usos: constantes, contadores, fábricas, utilidades (Math.max, Date.now, Object.keys... ¡son static!)



CLAVE — Pregunta para decidir: ¿necesita los datos de UNA instancia concreta (this)? → método normal. ¿No? → static.



Pasar un método como callback le quita su this

temporizador.js

```
class Temporizador {  
  segundos = 0;  
  tic() { this.segundos++; }  
}  
const t = new Temporizador();  
const f = t.tic;  
f(); // x TypeError  
setInterval(t.tic, 1000); // x no cuenta
```

¿Por qué?

t.tic

sin llamar () → solo la función, sin «t.»

Quien la llame después decide this: f() → undefined (TypeError). setInterval → window: suma en window.segundos (NaN) y t.segundos sigue a 0. Falla en silencio.

Pasa igual con forEach/map y con addEventListener (UD5).

1 Flecha al pasarla

```
setInterval(() => t.tic(), 1000);
```

2 bind

```
setInterval(t.tic.bind(t), 1000);
```

3 Campo flecha en la clase

```
tic = () => { this.segundos++; };
```



CLAVE — La más habitual es la 1. La 3 crea una función por instancia (lo que evitábamos en el bloque 2): úsala solo para manejadores de eventos.



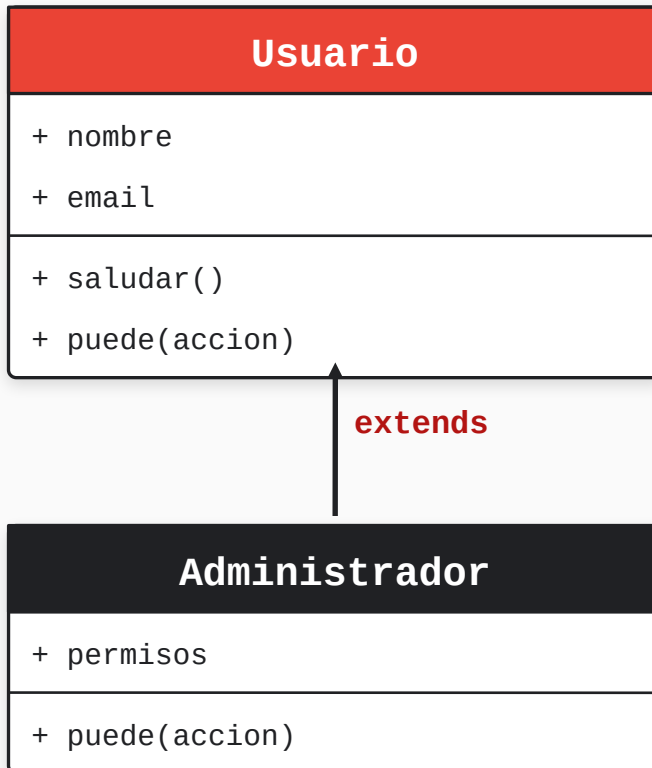
BLOQUE 4

Herencia y composición

extends, super, sobrescritura, la cadena con clases y cuándo NO heredar



Administrador ES UN Usuario, con algo más



```
usuarios.js

class Usuario {
  constructor(nombre, email) {
    this.nombre = nombre;
    this.email = email;
  }
  saludar() { return `Hola, ${this.nombre}`; }
  puede(accion) { return accion === "leer"; }
}

class Administrador extends Usuario {
  constructor(nombre, email, permisos) {
    super(nombre, email); // constructor del padre
    this.permisos = permisos;
  }
  puede(accion) { return this.permisos.includes(accion); }
}

const admin = new Administrador("Eva", "e@x.es", ["leer", "borrar"]);
admin.saludar(); // heredado: "Hola, Eva"
admin.puede("borrar"); // propio: true
```



CLAVE — La subclase hereda TODO lo público del padre y puede añadir (permisos) o cambiar (puede) comportamiento.



En una subclase, super() va antes de cualquier this

`new Administrador(...)`



constructor de Administrador



`super(nombre, email)`



Usuario crea el objeto (this)



ya existe this → `this.permisos = ...`

mal.js

```
class Administrador extends Usuario {  
  constructor(nombre, email, permisos) {  
    this.permisos = permisos; // x antes de super  
    super(nombre, email);  
  }  
}
```

ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor

- Si la subclase no escribe constructor, JS pone uno que llama a `super(...args)`
- Si escribe constructor, DEBE llamar a `super()` (si no: ReferenceError)



CLAVE — Es una regla del lenguaje, no una convención: el objeto lo construye la clase base, la subclase lo completa después.



Mismo mensaje, respuesta distinta según el objeto

figuras.js

```
class Figura {  
  area() { return 0; }  
  describir() {  
    return `${this.constructor.name}: ${this.area().toFixed(2)}`;  
  }  
}  
  
class Circulo extends Figura {  
  constructor(r) { super(); this.r = r; }  
  area() { return Math.PI * this.r ** 2; }  
}  
  
class Rectangulo extends Figura {  
  constructor(a, b) { super(); this.a = a; this.b = b; }  
  area() { return this.a * this.b; }  
}  
  
const figuras = [new Circulo(1), new Rectangulo(2, 3)];  
figuras.map(f => f.describir());  
// ["Circulo: 3.14", "Rectangulo: 6.00"]
```

f.area()

Circulo

$$\pi \cdot r^2$$

Rectangulo

$$a \cdot b$$

Sobrescribir: la subclase define un método con el mismo nombre.

super.metodo(): reutiliza la versión del padre y la amplía.

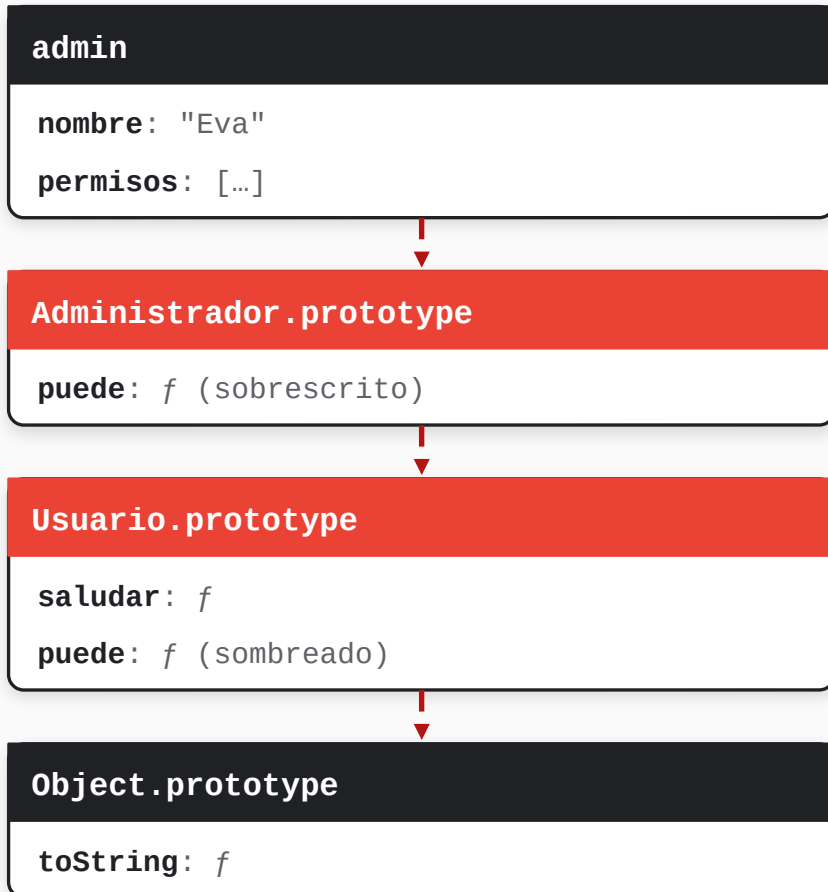
Polimorfismo: el código que recorre figuras no pregunta qué tipo es cada una; cada una sabe calcular su área.



CLAVE — Si te descubres escribiendo `if (tipo === "circulo") ... else if (tipo === "rectangulo") ...`, probablemente quieres polimorfismo.



extends enlaza prototipos: es la misma cadena del bloque 2



`admin.saludar()` → no está en `admin` → no está en `Administrador.prototype` → está en `Usuario.prototype` ✓

```
consola

admin instanceof Administrador; // true
admin instanceof Usuario;      // true
admin instanceof Object;       // true
Object.getPrototypeOf(Administrador.prototype)
=== Usuario.prototype;         // true
```

`instanceof` no mira «de qué clase se creó»: mira si `Clase.prototype` aparece en algún punto de la cadena del objeto.



NOTA — Por eso se puede heredar de clases nativas: `class MiError extends Error`, `class MiLista extends Array...`



Herencia («es un») o composición («tiene un»)

Herencia • extends

Administrador ES UN Usuario

Relación	es un / es una
Acoplamiento	Fuerte: si cambia el padre, cambian los hijos
Riesgo	Jerarquías rígidas y profundas
Encaja si	La jerarquía es real, estable y poco profunda

Composición

Coche TIENE UN Motor

Relación	tiene un / usa un
Acoplamiento	Débil: las piezas se cambian
Riesgo	Algo más de código para conectar
Encaja si	Se combinan comportamientos o pueden cambiar



CLAVE — «Favorece la composición» no es «nunca heredes»: es preguntarse cada vez si la relación es de verdad «es un».



El coche no hereda del motor: tiene uno y le delega el trabajo

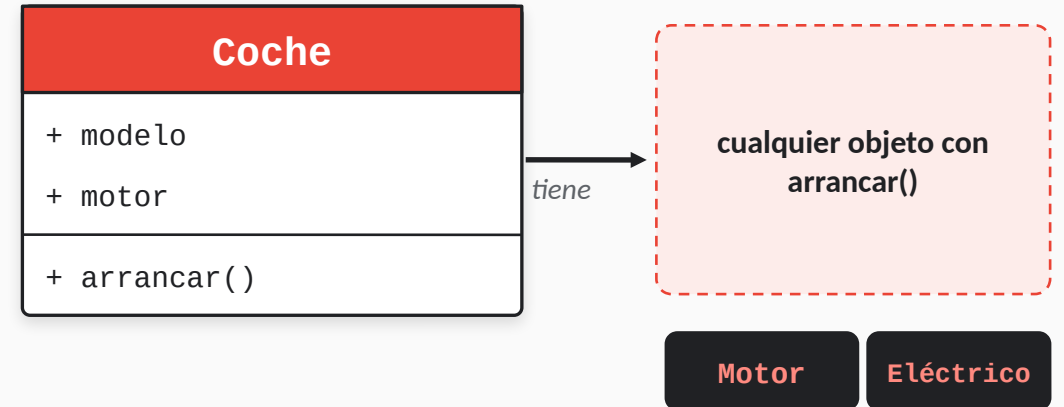
```
coche.js

class Motor {
  constructor(cv) { this.cv = cv; }
  arrancar() { return `Motor de ${this.cv} CV en marcha`; }
}

class MotorElectrico {
  arrancar() { return "Zumbido silencioso"; }
}

class Coche {
  constructor(modelo, motor) {
    this.modelo = modelo;
    this.motor = motor;      // TIENE UN motor
  }
  arrancar() {
    return `${this.modelo}: ${this.motor.arrancar()}`; // delega
  }
}

new Coche("León", new Motor(150)).arrancar();
new Coche("Born", new MotorElectrico()).arrancar();
```



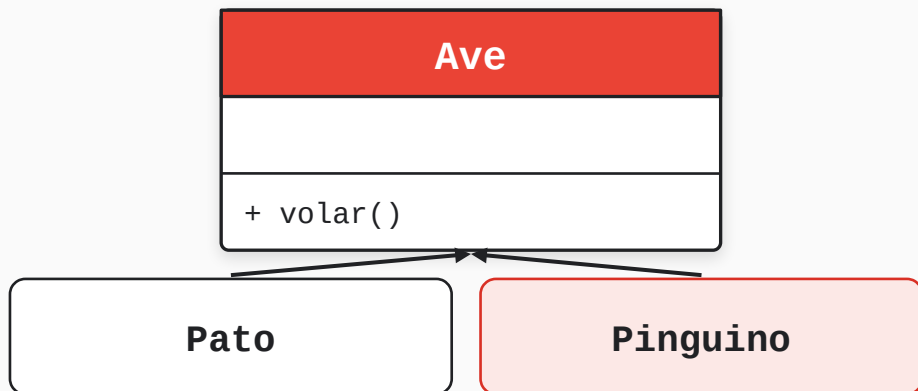
- Cambiar de motor = pasar otro objeto, sin tocar Coche
- Para probar Coche se le puede pasar un motor de mentira
- Con herencia harían falta CocheGasolina, CocheElectrico, CocheHibrido...



CLAVE — Delegar: el objeto no hace el trabajo él mismo, se lo pide a la pieza que tiene (this.motor.arrancar()).



El problema del pingüino: cuando la jerarquía no encaja



¿Pinguino.volar()? Hereda algo que no puede hacer. ¿Y nadar? ¿Pato y Puz comparten nadar pero no son de la misma familia?

```
capacidades.js

const puedeVolar = {
  volar() { return `${this.nombre} vuela`; },
};
const puedeNadar = {
  nadar() { return `${this.nombre} nada`; },
};

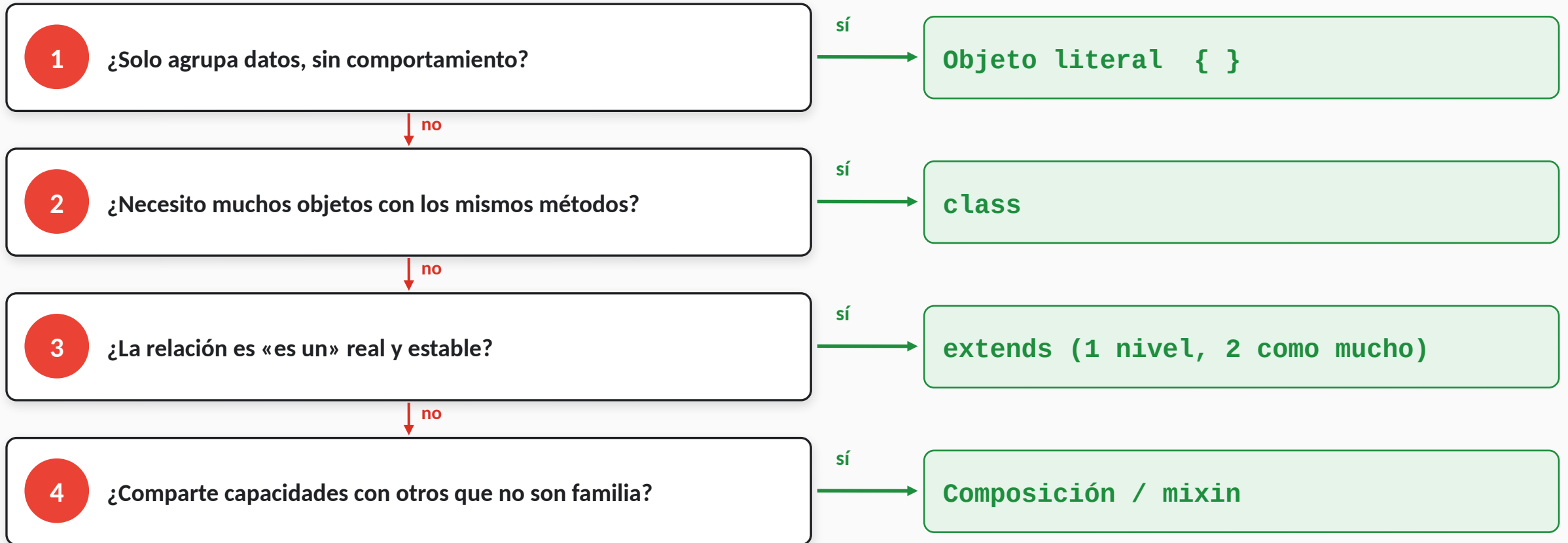
const pato = Object.assign({ nombre: "Pato" },
  puedeVolar, puedeNadar);
const pinguino = Object.assign({ nombre: "Pingu" },
  puedeNadar);
pinguino.nadar(); // "Pingu nada"
```

Mixin: objeto con métodos que se copian en otros (Object.assign). Cada animal recibe solo las capacidades que tiene. Los métodos usan this, así que funcionan en cualquier objeto que tenga nombre.

NOTA — Object.assign(destino, ...origenes) copia las propiedades propias de los orígenes en el destino y lo devuelve.



Cuatro preguntas antes de escribir class



CLAVE — Ninguna respuesta es obligatoria: es una conversación de equipo. Lo que no vale es heredar «para reutilizar código» sin relación real.



BLOQUE 5

Objetos nativos del lenguaje

Math, Date, String, Number, Intl y Error: no reinventes lo que ya existe



JavaScript trae objetos listos para usar



Math

Cálculos

`Math.round, Math.random`



Date

Fechas y horas

`new Date(), Date.now()`



String

Texto

`trim, includes, padStart`



Number

Números

`toFixed, Number.isNaN`



Array

Listas (UD3)

`map, filter, reduce`



Intl

Formato local

`moneda, fechas en es-ES`



JSON

Texto ↔ objeto

`stringify / parse (UD6)`



Error

Errores

`throw new Error(...)`

¿Por qué `"hola".toUpperCase()` funciona si `"hola"` es un primitivo? JS lo envuelve un instante en un objeto String y busca el método en `String.prototype`. Otra vez la cadena de prototipos.



NOTA — Math, JSON e Intl no se instancian con `new`: son objetos con métodos estáticos. Date, Error, Map... sí son clases.



Redondear, aleatorios y la trampa de los decimales

Valor	round	floor	ceil	trunc
2.5	3	2	3	2
-2.5	-2	-3	-2	-2
2.49	2	2	3	2

math.js

```
// entero aleatorio entre min y max (ambos incluidos)
const azar = (min, max) =>
  Math.floor(Math.random() * (max - min + 1)) + min;

Math.max(...[4, 9, 2]); // 9 (spread, UD3)
Math.min();             // Infinity (¡ojo!)
```

number.js

```
0.1 + 0.2; // 0.30000000000000004
(0.1 + 0.2).toFixed(2); // "0.30" ← string
Number("12.5"); // 12.5
Number("12,5"); // NaN (coma española)
Number.parseFloat("12.5€"); // 12.5
Number.isNaN(NaN); // true
NaN === NaN; // false
```

Dinero: redondea solo al MOSTRAR (toFixed / Intl) o trabaja en céntimos enteros (1999 en vez de 19.99). Nunca compares decimales con ===.



CLAVE — toFixed devuelve un STRING. Si después sumas, "0.30" + 1 = "0.301". Convierte de vuelta con Number() o espera a mostrar.



Fechas: la clase con más trampas del lenguaje

```
fechas.js

const ahora = new Date();
const examen = new Date(2026, 10, 27); // ¡27 de NOVIEMBRE!
const iso = new Date("2026-11-27T09:00"); // hora local
const utc = new Date("2026-11-27"); // ¡sin hora: UTC!

examen.getFullYear(); // 2026
examen.getMonth(); // 10 (0 = enero)
examen.getDate(); // 27 (día del mes)
examen.getDay(); // 5 (0 = domingo)

// diferencia en días
const ms = examen - ahora; // milisegundos
const dias = Math.ceil(ms / 86_400_000);

Date.now(); // ms desde 1-1-1970 (marca de tiempo)
```

Meses de 0 a 11

`new Date(2026, 10, 27)` es noviembre

`getDay` ≠ `getDate`

`getDay` = día de la semana; `getDate` = día del mes

Restar fechas da ms

1 día = 86 400 000 ms

Es mutable

`setDate()` cambia el objeto original

ISO sin hora = UTC

"2026-11-27" es medianoche UTC: en América, día 26

Para mostrarla: siguiente diapositiva (Intl).



NOTA — Temporal (la nueva API de fechas) ya está llegando a los navegadores; `Date` seguirá en todo el código existente durante años.



Limpiar texto y mostrar datos como se esperan en España

texto.js

```
const entrada = " Ana@Correo.ES ";
entrada.trim().toLowerCase(); // "ana@correo.es"
"Teclado".includes("cla"); // true
"7".padStart(3, "0"); // "007"
"a-b-c".split("-"); // ["a", "b", "c"]
"2026-11-27".slice(0, 4); // "2026"
"x.y.z".replaceAll(".", "/"); // "x/y/z"

// Los strings son inmutables:
entrada.trim(); // devuelve uno NUEVO
entrada; // sigue con espacios
```

intl.js

```
const eur = new Intl.NumberFormat("es-ES",
  { style: "currency", currency: "EUR" });
eur.format(1234.5); // "1234,50 €"

examen.toLocaleDateString("es-ES",
  { weekday: "long", day: "numeric", month: "long" });
// "viernes, 27 de noviembre"
```

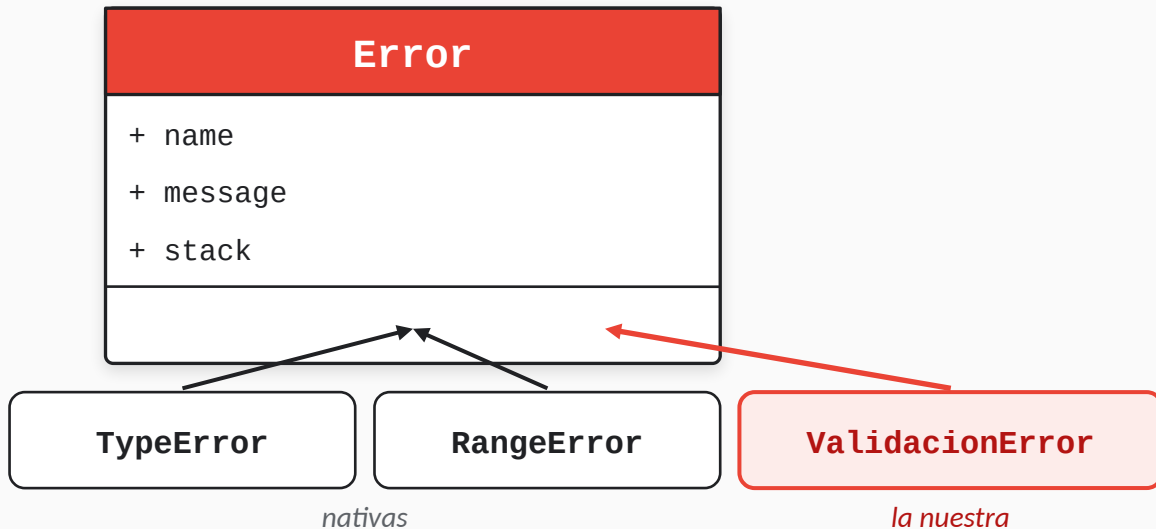
Intl es una API del lenguaje: formatea moneda, fechas, números y plurales según el idioma. Nada de concatenar "€" ni de cambiar puntos por comas a mano.



CLAVE — Ningún método de String modifica el original: siempre devuelven un string nuevo que hay que guardar.



Error es una clase... y se puede heredar de ella



- Herencia bien usada: un ValidacionError ES UN Error
- instanceof distingue los errores esperados de los bugs
- Lanzad siempre new Error(...), nunca throw "texto" (sin stack)

```
errores.js

class ValidacionError extends Error {
  constructor(campo, mensaje) {
    super(mensaje);
    this.name = "ValidacionError";
    this.campo = campo;
  }
}

try {
  throw new ValidacionError("email", "Email no válido");
} catch (e) {
  if (e instanceof ValidacionError) {
    console.warn(e.campo, e.message);
  } else throw e;
}
```



CLAVE — Aquí se junta todo: clase nativa + extends + super + instanceof. Lo usaréis en los formularios (UD5) y en fetch (UD6).



BLOQUE 6

Manos a la obra

Laboratorio, DevTools, caso real, errores comunes, control y mini-reto



Modelo de una tienda con clases (sin DOM todavía)



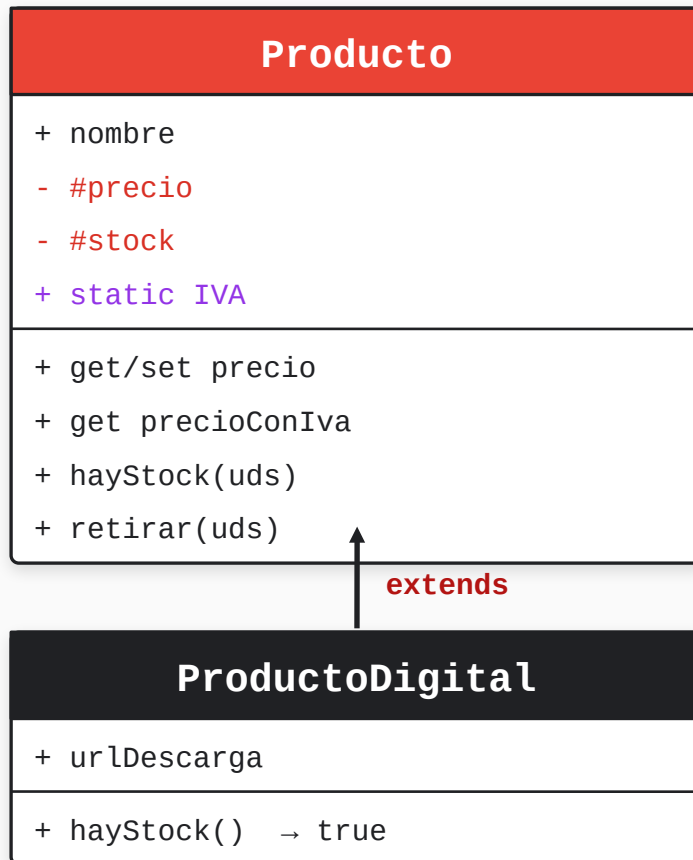
Proyecto Vite (UD1) con un fichero por clase en `src/modelo/` y un `main.js` que solo crea objetos y hace `console.log` / `console.table`. Es la base del modelo de la Práctica 2.



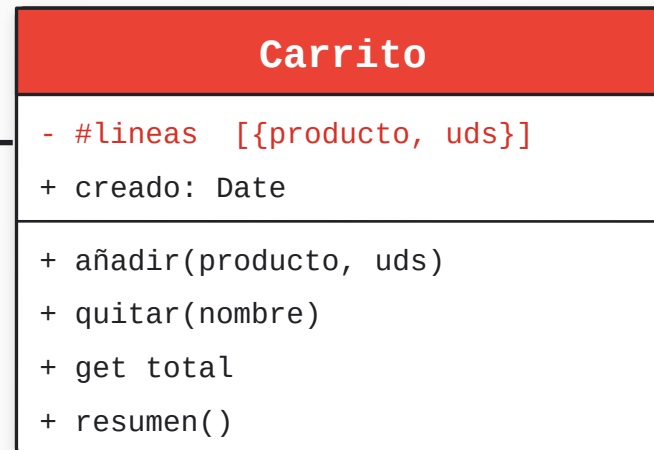
ÉXITO — Criterio de éxito: precio negativo → `RangeError`, `#stock` inaccesible desde `main.js`, total en formato «129,98 €» y Carrito no hereda de nadie.



El diagrama que tenéis que convertir en código



tiene 0..*



Si acabáis: un descuento como pieza intercambiable (composición) que el Carrito usa al calcular el total.

main.js

```
const c = new Carrito();
c.añadir(teclado, 2);
c.añadir(ebook, 1);
c.total; // 129.98
c.resumen();
// "3 artículos · 129,98 €"

// cada error, en su try/catch
try { teclado.precio = -1; }
catch (e) {...} // RangeError
try { c.añadir(raton, 99); }
catch (e) {...} // sin stock
```



NOTA — añadir llama a producto.retirar(uds). ¿c.total da 129.98000000000002? Decimales (bloque 5): redondea en get total.



Los objetos se inspeccionan, no se adivinan

```
console.log(obj)
```

Vista rápida; se despliega con ►. Ojo: muestra el estado al DESPLEGAR, no al imprimir

```
console.dir(obj)
```

Vista como objeto, con `[[Prototype]]` desplegable

```
console.table(lista)
```

Array de objetos en forma de tabla: ideal para el carrito

```
Object.getPrototypeOf(x)
```

Subir un eslabón de la cadena a mano

```
debugger;
```

Pausa en esa línea con DevTools abierto: ver this y las variables

Ojo: Chrome rotula cada `[[Prototype]]` con el nombre de SU padre. El primer «`[[Prototype]]: Usuario`» es en realidad `Administrador.prototype` (dentro se ve `constructor: class Administrador`).

```
> console.dir(admin)
▼ Administrador
  nombre: "Eva"
  email: "e@x.es"
  permisos: (2) ['leer', 'borrar']
  ▼ [[Prototype]]: Usuario
    constructor: class Administrador
    puede: f puede(accion)
    ▼ [[Prototype]]: Object
      saludar: f saludar()
      puede: f puede(accion)
      ► [[Prototype]]: Object
```



NOTA — En la consola podéis leer `#privados` para depurar. En el código, no. Que funcione en la consola no prueba nada.



CASO REAL

Dónde os vais a encontrar todo esto



El propio navegador

HTMLButtonElement → HTMLElement → Element → Node → EventTarget. El DOM de la UD5 es una cadena de prototipos.



Web Components

class MiBoton extends HTMLElement { ... } — componentes nativos reutilizables.



React antes de 2019

class Contador extends React.Component. Hoy se usan funciones y hooks: lo veremos en la UD7 y entenderéis por qué.



TypeScript

Añade interfaces, tipos y private/protected en tiempo de compilación sobre estas mismas clases.



Backend y librerías

Modelos (NestJS, Angular, ORM), errores propios, SDKs de APIs: todo clases.



Tendencia actual

Se mezcla POO y programación funcional según convenga. Sin dogmas.



DATO — Probadlo hoy: `Object.getPrototypeOf(document.body)` en la consola y seguid subiendo.



ERRORES COMUNES (1/2)

Clases y herencia

Olvidar new

La clase no se puede llamar como función

❌ `Producto("Ratón", 15)`

✅ `new Producto("Ratón", 15)`

this antes de super()

ReferenceError en el constructor de la subclase

❌ `this.permisos = p; super(n, e);`

✅ `super(n, e); this.permisos = p;`

Heredar para reutilizar

Carrito no ES UN Array ni un Producto

❌ `class Carrito extends Producto`

✅ `class Carrito { #lineas = []; }`

Sobrescribir y perder al padre

El método nuevo olvida lo que hacía el original

❌ `describir() { return "Admin"; }`

✅ `describir() { return super.describir() + "!"; }`



this, encapsulación y nativos

Método flecha en un objeto

this no es el objeto

✗ `saludar: () => this.nombre`

✓ `saludar() { return this.nombre; }`

Pasar el método sin su objeto

this se pierde en el callback

✗ `lista.forEach(t.tic)`

✓ `lista.forEach(() => t.tic())`

_privado que no es privado

Cualquiera puede escribir `cuenta._saldo = 1e6`

✗ `this._saldo = 0;`

✓ `#saldo = 0;`

Meses de Date desde 1

Sale el mes siguiente

✗ `new Date(2026, 11, 27) // ¿nov?`

✓ `new Date(2026, 10, 27) // nov`



Preguntas de control y debate

- 1 ¿Por qué se dice que class es azúcar sintáctico? ¿Qué hay debajo?
- 2 Si p1 y p2 son instancias de Persona, ¿cuántas copias de saludar existen? ¿Dónde?
- 3 ¿Qué diferencia real hay entre #saldo y _saldo? ¿Qué error da cuenta.#saldo fuera de la clase?
- 4 setInterval(t.tic, 1000) no cuenta (t.segundos sigue a 0). ¿Por qué? Da dos arreglos.
- 5 Carrito y Producto: ¿herencia o composición? Justifícalo con «es un» / «tiene un».



MINI-RETO PRÁCTICO

Reto express: 20 minutos, en parejas

1

Vehiculo

marca, modelo, #km con get y método conducir(km)

2

Setter que valida

km negativos → RangeError

3

CocheElectrico

extends Vehiculo; super() bien; autonomía

4

Composición

una Bateria aparte que el coche TIENE

5

Probar el privado

intentar leer #km desde fuera

Revisión cruzada

A los 15 min, intercambiad el código y comprobad:



Pareja A



Pareja B

- ¿#km es inaccesible fuera de la clase?
- ¿conducir(-5) lanza RangeError?
- ¿Bateria no hereda de nadie?



ÉXITO — Criterio de éxito: el SyntaxError al leer #km fuera de la clase, el RangeError del setter y la batería como objeto aparte.



Objetos y prototipos

objeto

Colección de propiedades clave → valor

propiedad

Par clave-valor de un objeto

método

Propiedad cuyo valor es una función

this

Objeto sobre el que se llama el método (izquierda del punto)

referencia

La variable apunta al objeto; no lo contiene

copia superficial

{ ...obj }: copia 1 nivel; lo anidado se comparte

[[Prototype]]

Enlace interno de un objeto a su prototipo

cadena de prototipos

Prototipo del prototipo... hasta null

constructora

Función que se llama con new para crear objetos

Object.create

Crea un objeto con el prototipo indicado

azúcar sintáctico

Sintaxis más cómoda para algo que ya existía

Object.freeze

Impide cambiar el objeto (solo 1.er nivel)



Clases y diseño

clase

Plantilla para crear objetos con los mismos métodos

instancia

Objeto concreto creado con `new Clase()`

constructor

Método que inicializa la instancia

get / set

Métodos que se usan como si fueran propiedades

#privado

Campo o método accesible solo dentro de la clase

encapsulación

Ocultar el estado y exponer solo una interfaz

static

Miembro de la clase, no de cada instancia

extends / super

Heredar de una clase / llamar al padre

sobrescribir

Redefinir en la subclase un método heredado

polimorfismo

Mismo método, distinta respuesta según el objeto

composición

Un objeto TIENE otros y les delega trabajo

mixin

Objeto de métodos que se copian en otros



La POO es una herramienta, no una religión



No todo es una clase

Un objeto literal o una función a veces es mejor. Si una clase solo tiene constructor y datos, quizá sobra.



El debate sigue vivo

Herencia o composición se discute en cada proyecto real, también entre gente con mucha experiencia.



Decisión consciente

No apliques un patrón «porque toca»: aplícalo porque encaja en el problema y lo puedes justificar.



Pregunta para casa: piensa en una app que uses a diario. Un caso donde heredar tenga sentido y otro donde la composición encaje mejor.

CHECKPOINT

Resumen de puntos clave



Debajo de class hay funciones y una cadena de prototipos



Datos en cada instancia; métodos en el prototipo, compartidos



En una subclase, super() antes de usar this



#privado lo hace cumplir el lenguaje; _privado es solo un aviso



Composición o herencia: decisión consciente, «tiene un» frente a «es un»



Math, Date, String e Intl antes que reinventar

SIGUIENTE • UD5

DOM nativo y gestión de eventos

- Pintar vuestro modelo en la página
- addEventListener y el this perdido
- Práctica 2: clases + DOM + eventos

Hoy: el modelo. La semana que viene: que se vea y responda en el navegador.