

JavaScript para Profesionales

Fundamentos

Objetos

- Conjunto de propiedades propias + heredadas de otro objeto (prototipos)
- ¡Qué no cunda el pánico!

Objetos

- Dinámicos

```
var obj = {};
obj.nuevaPropiedad = 1;
delete obj.nuevaPropiedad;
```

- Set de strings

```
var strset = {
  hola: true,
  adios: true
};
"holo" in strset;
```

Objetos

- Referencias

```
var p1 = {x: 1},  
      p2 = p1;
```

```
p1 === p2; // true  
p1.x = 5;  
p2.x; // 5
```

- Todo son objetos excepto: strings, números, booleans, `null` o `undefined`
- Strings, números y booleans se comportan como objetos inmutables

Objetos

- Ciudadanos de primer orden

```
(function (obj) {  
    return {b: 2};  
})({a: 1});
```

- Contener valor primitivo u otros objetos. Incluyendo funciones.

```
var obj = {  
    f: function() {  
        console.log("holá");  
    }  
};  
obj.f();
```

Objetos

- Literales
 - clase: Object
 - sencillos y ligeros
 - Construidos
 - clase: **prototype**
 - constructor
- ```
{
 un: "objeto",
 literal: true
};
```
- new **NombreDeUnaFuncion()**;

# Clases

- Pueden entenderse como:
  - Tipo (jerárquico) de datos
    - Aquí no
  - Categoría de objetos con la misma estructura
  - Al grano: objetos con el mismo prototipo.

# Clases

Si esto es un “punto”

```
var point = {x: 0, y: 0};
```

Y esto es otro “punto”

```
var point2 = {x: 5, y: 5};
```

¿Qué es esto?

```
var what = {x: 10, y: 10};
```

¿Y esto?

```
var isit = {x: -10, y: -20};
```

# Mensajes

- Teniendo:

```
var obj = {
 nombre: "Pepito",
 saludo: function () {
 return "Hola, Mundo!";
 }
};
```

- ¿Que significa esto?

```
obj.nombre;
```

# Mensajes

- Teniendo:

```
var obj = {
 nombre: "Pepito",
 saludo: function () {
 return "Hola, Mundo!";
 }
};
```

- ¿Y esto?

```
obj.saludo;
```

# Mensajes

- Teniendo:

```
var obj = {
 nombre: "Pepito",
 saludo: function () {
 return "Hola, Mundo!";
 }
};
```

- ¿Y esto otro? ¡cuidado!

```
obj["saludo"]();
```

# Mensajes

- Teniendo:

```
var obj = {
 nombre: "Pepito",
 saludo: function () {
 return "Hola, Mundo!";
 }
};
```

- ¿Es lo mismo?

```
var fn = obj["saludo"];
fn();
```

# Mensajes

- Teniendo:

```
var obj = {
 nombre: "Pepito",
 saludo: function () {
 return "Hola, Mundo!";
 }
};
```

- ¡NO es no mismo!

```
var fn = obj["saludo"];
fn();
```

# Mensajes

- Una función se puede ejecutar de 4 maneras:

- Invocando directamente la función

```
(function() { alert("Hey!"); })();
```

- Enviando un mensaje a un objeto (método)

```
objeto.metodo();
```

- Como constructor

```
new MiConstructor();
```

- Indirectamente, a través de `call(...)` y `apply(...)`

```
fn.call({}, "param");
```

# Mensajes

- Invocando directamente la función

```
(function() { alert("Hey!"); })();
```

- Enviando un mensaje a un objeto (método)

```
objeto.metodo();
```

# Mensajes

Un mensaje se envía a un receptor

```
var obj = {};
obj.toString(); // [object Object]

"hola don Pepito".toUpperCase();
```

# Mensajes

Un mensaje se envía a un receptor

```
var obj = {};
obj.toString(); // [object Object]

"hola don Pepito".toUpperCase();
```

# Mensajes

La sintaxis es engañosa

```
var obj = {
 colección: ["uno", "dos", "tres"],
 método: function() {
 return "Hola, Mundo!";
 }
};
```

obj.colección[1];                            vs.                            obj.método();

# Mensajes

```
var fn = obj.metodo;
fn();
```

obj.metodo();

# Mensajes

```
var fn = obj.metodo;
fn();
obj.metodo();
```

- Accede a la propiedad “metodo” de **obj**
- Supongo que es una función y la invoco

# Mensajes

```
var fn = obj.metodo;
fn();
```

```
obj.metodo();
```

- Accede a la propiedad “metodo” de **obj**
- Supongo que es una función y la invoco
- Envía el mensaje “metodo” a **obj**
- Si existe, **obj** se encarga de ejecutar la función

# Mensajes

```
var fn = obj.metodo;
fn();
```

```
obj.metodo();
```

- Accede a la propiedad “metodo” de **obj**
- Supongo que es una función y la invoco
- NO HAY RECEPTOR
- Envía el mensaje “metodo” a **obj**
- Si existe, **obj** se encarga de ejecutar la función
- **obj** ES EL RECEPTOR

# Mensajes

Un error típico:

```
$("#elemento").click(objeto.clickHandler);
```

# Mensajes

Un error típico:

```
$("#elemento").click(objeto.clickHandler);
```

- Lo que se intenta decir:
  - “Al hacer click sobre #elemento, envía el mensaje clickHandler a objeto”

# Mensajes

Un error típico:

```
$("#elemento").click(objeto.clickHandler);
```

- Lo que se intenta decir:
  - “Al hacer click sobre `#elemento`, envía el mensaje `clickHandler` a `objeto`”
- Lo que se dice en realidad:
  - “Accede al valor de la propiedad `clickHandler` de `objeto` y ejecútalo al hacer click sobre `#elemento`”

# **El receptor: ...**

¿Por qué tanto lío con el receptor del mensaje?

# El receptor: **this**

¿Por qué tanto lío con el receptor del mensaje?

- ¡El receptor es **this**!
- La metáfora mensaje/receptor aclara su (escurridizo) significado

# El receptor: this

this = “el receptor de este mensaje”

```
var nombre = "Sonia";

var obj = {
 nombre: "Pepito",
 saludo: function() {
 alert("hola " + this.nombre);
 }
}

obj.saludo();
```

# **El receptor: this**

**this**

- Su significado es dinámico
- Se decide en el momento (y según la manera) de ejecutar la función
- Se suele llamar “el contexto de la función”
- Cuando no hay receptor, apunta al objeto global

# El receptor: this

Cuando no hay receptor, es el objeto global

```
var nombre = "Sonia"

var obj = {
 nombre: "Pepito",
 saludo: function() {
 alert("hola " + this.nombre)
 }
}

var fn = obj["saludo"];
fn();
```

# El receptor: this

Su valor es dinámico

```
var obj = {
 nombre: "Pepito",
 saludo: function() {
 alert("hola " + this.nombre);
 }
};
```

```
var maria = {
 nombre: "María"
};
```

```
maria.saludo = obj.saludo;
maria.saludo();
```

# El receptor: this

Semánticamente, es como un parámetro oculto

```
function ([this]) {
 alert("hola " + this.nombre);
}
```

que el receptor se encargara de proveer

```
obj.saludo(); => saludo([obj]);
```

# El receptor: this

Semánticamente, es como un parámetro oculto

```
var nombre = "Sonia";

var obj = {
 nombre: "Pepito",
 saludo: function() {
 var saludo_fn = function() {
 alert("hola " + this.nombre);
 };
 saludo_fn();
 };
};

obj.saludo();
```

# El receptor: this

Semánticamente, es como un parámetro oculto

```
var nombre = "Sonia";

var obj = {
 nombre: "Pepito",
 saludo: function([this]) {
 var saludo_fn = function([this]) {
 alert("hola " + this.nombre);
 };
 saludo_fn([objeto global]);
 };
};

obj.saludo([obj]);
```

# El receptor: this

Semánticamente, es como un parámetro oculto

```
var nombre = "Sonia";

var obj = {
 nombre: "Pepito",
 saludo: function([this]) {
 var saludo_fn = function([this]) {
 alert("hola " + [this].nombre);
 };
 saludo_fn([objeto global]);
 };
};

obj.saludo([obj]);
```

# El receptor: **this**

Es decir:

- Cada función tiene su propio **this**
- Una función anidada en otra NO comparte el receptor
- El valor de **this** depende de la invocación, NO de la definición (no se clausura)

# El receptor: this

Otro error común:

```
var obj = {
 clicks: 0,
 init: function() {
 $("#element").click(function() {
 this.clicks += 1;
 });
 }
};

obj.init();
```

# El receptor: this

Otro error común:

```
var obj = {
 clicks: 0,
 init: function(this) {
 $("#element").click(function(this) {
 this.clicks += 1;
 });
 }
};

obj.init(obj);
```

# El receptor: this

Una posible solución (aunque no la mejor):

```
var obj = {
 clicks: 0,
 init: function() {
 var that = this;
 $("#element").click(function() {
 that.clicks += 1;
 });
 }
};

obj.init();
```



# Repasso: Mensajes

- Una función se puede ejecutar de 4 maneras:

- Invocando directamente la función

```
(function() { alert("Hey!"); })();
```

- Enviando un mensaje a un objeto (método)

```
objeto.metodo();
```

- Como constructor

```
new MiConstructor();
```

- Indirectamente, a través de `call(...)` y `apply(...)`

```
fn.call({}, "param");
```

# Reaso: Mensajes

- Indirectamente, a través de `call(...)` y `apply(...)`

```
fn.call([], "param");
```

# El receptor: this

- Las funciones son objetos
- Se pueden manipular como cualquier otro objeto
  - Asignar valores a propiedades
  - Pasar como parámetros a otras funciones
  - Ser el valor de retorno
  - Guardarse en variables u otros objetos
- Tienen métodos

```
var fn = function() { alert("Hey!"); };

fn.toString();
```

# El receptor: this

- Dos métodos permiten manipular el receptor (contexto):

- fn.call(context [, arg1 [, arg2 [...]]])

```
var a = [1,2,3];
Array.prototype.slice.call(a, 1, 2); // [2]
```

- fn.apply(context, arglist)

```
var a = [1,2,3];
Array.prototype.slice.apply(a, [1, 2]); // [2]
```

# El receptor: this

```
var nombre = "Objeto Global";

function saluda() {
 alert("Hola! Soy " + this.nombre);
}

var alicia = {
 nombre: "Alicia"
};

saluda();

saluda.call(alicia);
```

# arguments

- El otro parámetro oculto
- Contiene una lista de todos los argumentos
- NO es un Array

```
function echoArgs() {
 alert(arguments); // [object Arguments]
}

echoArgs(1, 2, 3, 4);
```

# arguments

- Se comporta (más o menos) como Array...

```
function echoArgs() {
 alert(arguments[0]); // 1
}
```

```
echoArgs(1, 2, 3, 4);
```

- ...pero NO del todo

```
function echoArgs() {
 return arguments.slice(0, 1); // Error!
}
```

```
echoArgs(1, 2, 3, 4);
```

# arguments

- Un truco:

```
function echoArgs() {
 var slice = Array.prototype.slice;
 return slice.call(arguments, 0, 1);
}

echoArgs(1, 2, 3, 4); // [1]
```

# arguments

¡Cuidado, se comporta como parámetro oculto!

```
function exterior() {
 var interior = function() {
 alert(arguments.length);
 };
 interior();
}

exterior("a", "b", "c");
```

# intermedio: this y arguments

¿Qué hace esta función?

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var algo = misterio();

typeof algo; // ???
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var algo = misterio();

algo(); // ???
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var algo = misterio({}, function() {
 return this;
});

typeof algo(); // ???
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var obj = {};

var algo = misterio(obj, function() {
 return this;
});

obj === algo(); // ???
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var obj = {};

var algo = misterio({}, function() {
 return this;
});

obj === algo(); // ???
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var obj = {
 nombre: "Bárbara"
};

var algo = misterio(obj, function() {
 return this.nombre;
});

algo(); //?
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var obj = {
 nombre: "Bárbara"
};

var algo = misterio(obj, function (saludo) {
 return saludo + " " + this.nombre;
});

algo("Hola, "); //???
```

# Intermedio: this y arguments

```
function misterio(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}

var barbara = { nombre: "Bárbara" };
var carlos = { nombre: "Carlos" };

var algo = misterio(barbara, function (saludo) {
 return saludo + " " + this.nombre;
});

algo.call(carlos, "Hola, "); //???
```

# Intermedio: this y arguments

- `bind`: fija una función a un contexto

```
function bind(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}
```

# Intermedio: this y arguments

Volviendo al problema:

```
var obj = {
 clicks: 0,
 init: function() {
 $("#element").click(function() {
 // MAL
 this.clicks += 1;
 });
 }
};

obj.init();
```

# Intermedio: this y arguments

Apaño:

```
var obj = {
 clicks: 0,
 init: function() {
 var that = this;
 $("#element").click(function() {
 that.clicks += 1;
 });
 }
};

obj.init();
```

# Intermedio: this y arguments

¿Qué pasa si el callback es un método?

```
var obj = {
 clicks: 0,
 incClicks: function() {
 this.clicks += 1;
 },
 init: function() {
 $("#element").click(
 // ???
);
 }
};

obj.init();
```

# Intermedio: this y arguments

Apaño cada vez más feo:

```
var obj = {
 clicks: 0,
 incClicks: function() {
 this.clicks += 1;
 },
 init: function() {
 var that = this;
 $("#element").click(function() {
 that.incClicks();
 });
 }
};
```



# Intermedio: this y arguments

- bind al rescate

```
var obj = {
 clicks: 0,
 incClicks: function() {
 this.clicks += 1;
 },
 init: function() {
 $("#element").click(
 bind(this, this.incClicks)
);
 }
};
```

# intermedio: this y arguments

¿Qué hace esta otra función?

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

typeof enigma(); // ???
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

var cosa = enigma();
typeof cosa(); // ???
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

var cosa = enigma(function() {
 return "Hola!";
});

cosa(); // ???
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

function saluda(nombre) {
 return "Hola, " + nombre + "!";
}

var cosa = enigma(saluda);

cosa("Mundo"); // ???
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

function saluda(nombre) {
 return "Hola, " + nombre + "!";
}

var cosa = enigma(saluda, "Mundo");

cosa(); // ???
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

function saluda(saludo, nombre) {
 return saludo + ", " + nombre + "!";
}

var cosa = enigma(saluda, "Hola", "Mundo");

cosa(); // ???
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

function saluda(saludo, nombre) {
 return saludo + ", " + nombre + "!";
}

var cosa = enigma(saluda, "Hola");

cosa("Mundo"); // ???
cosa("Don Pepito"); // ???
```

# intermedio: this y arguments

```
function enigma(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}

var dario = {nombre: "Dario"};
var elena = {nombre: "Elena"};

function saluda(saludo) {
 return saludo + ", " + this.nombre + "!";
}

var cosa = enigma(saluda, "Qué pasa");

cosa.call(dario); // ???
cosa.call(elena); // ???
```

# Intermedio: this y arguments

- **curry**: aplicación parcial de una función

```
function curry(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, args.concat(newargs));
 };
}
```

# Intermedio: rizar el rizo

```
var unObj = {
 nombre: "Manuel",
 edad: 32
};

function getNombre() { return this.nombre; }
function setNombre(nombre) { this.nombre = nombre; }
function getEdad() { return this.edad; }
function setEdad(edad) { this.edad = edad; }

var bindToUnObj = curry(bind, unObj),
 getUnObjNombre = bindToUnObj(getNombre),
 setUnObjNombre = bindToUnObj(setNombre);

setUnObjNombre("Pepito");
getUnObjNombre(); // ???
```

# Intermedio: rizar el rizo

```
function getter(prop) { return this[prop]; }
function setter(prop, value) { this[prop] = value; }

var manuel = {
 nombre: "Manuel",
 edad: 32
};

var edadDeManuel = bind(manuel, curry(getter, "edad"));
edadDeManuel(); // ???
```

# **Prototipos**

# Prototipos

- ¿Por qué tan mala fama?
- ¡Es un mecanismo muy sencillo!
- Distinto a otros lenguajes

# Prototipos

Un objeto `obj`:

```
var obj = {uno: 1, dos: 2};
```

qué pasa si hacemos:

```
obj.uno; // 1
```

# Prototipos

```
var obj = {uno: 1, dos: 2};
```

| obj |   |
|-----|---|
| uno | 1 |
| dos | 2 |

```
obj.uno; // 1
```



| obj |   |
|-----|---|
| uno | 1 |
| dos | 2 |

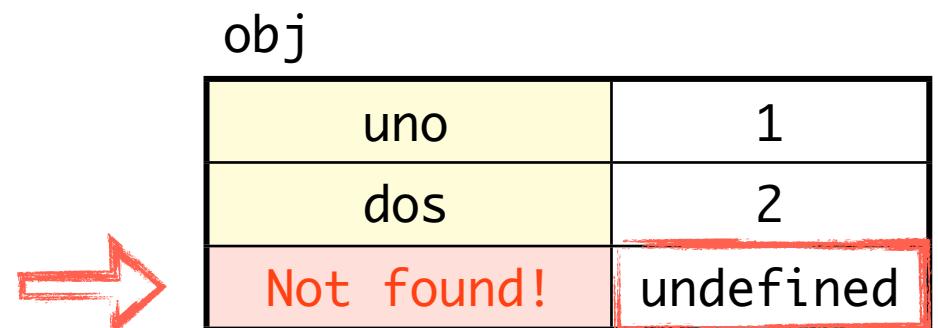
# Prototipos

```
var obj = {uno: 1, dos: 2};
```

| obj |   |
|-----|---|
| uno | 1 |
| dos | 2 |

Si hacemos:

```
obj.tres; // undefined
```



# Prototipos

```
var obj = {uno: 1, dos: 2};
```

| obj |   |
|-----|---|
| uno | 1 |
| dos | 2 |

¿De dónde sale?

```
obj.toString(); // '[object Object]'
```

| obj        |           |
|------------|-----------|
| uno        | 1         |
| dos        | 2         |
| Not found! | undefined |



?

# Prototipos

```
obj.toString(); // '[object Object]'
```

obj

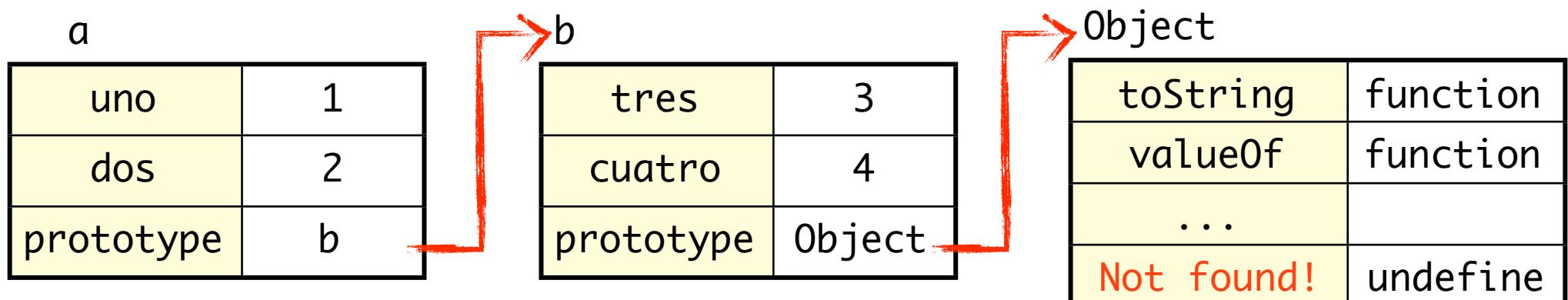
|           |        |
|-----------|--------|
| uno       | 1      |
| dos       | 2      |
| prototype | Object |

Object

|            |           |
|------------|-----------|
| toString   | function  |
| valueOf    | function  |
| ...        |           |
| Not found! | undefined |

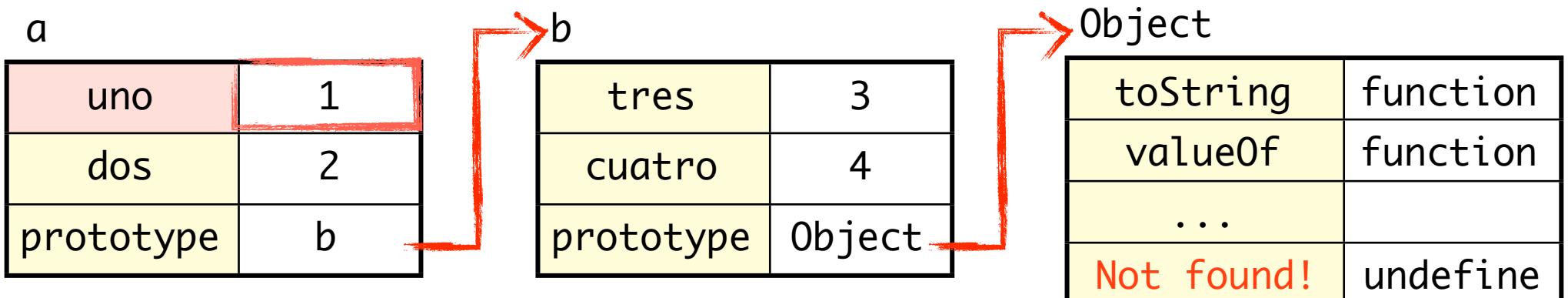
# Prototipos

Teniendo:



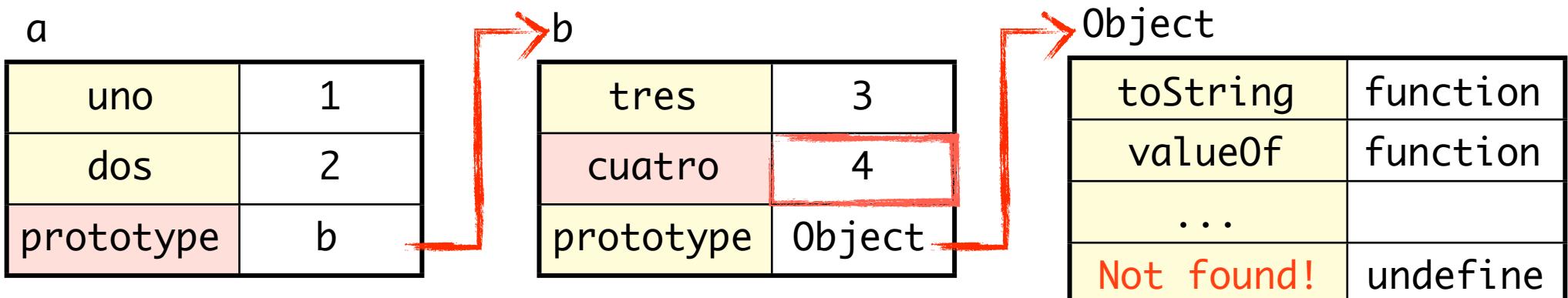
# Prototipos

a.uno; // 1



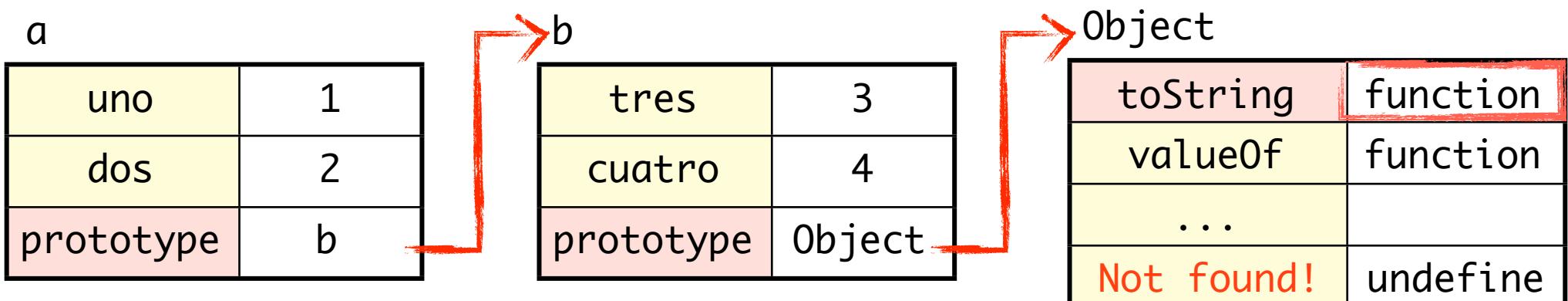
# Prototipos

```
a.cuatro; // 4
```



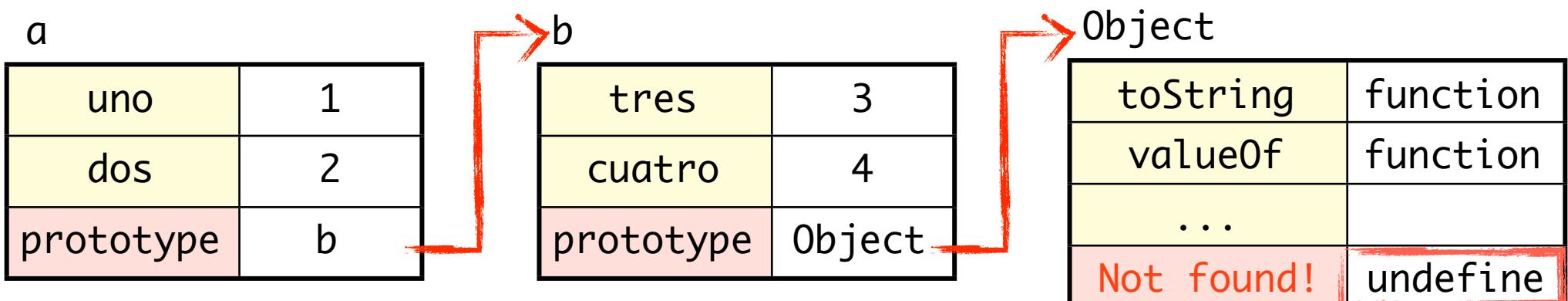
# Prototipos

```
a.toString; // [object Object]
```



# Prototipos

```
a.noExiste; // undefined
```



# Prototipos

Pero... ¿Cómo establezco el prototipo de un objeto?

- No se puede hacer directamente
- No se puede modificar el prototipo de objetos literales
- Solo objetos generados (con `new`)
- Constructores!

# Repasso: Mensajes

- Una función se puede ejecutar de 4 maneras:

- Invocando directamente la función

```
(function() { alert("Hey!"); })();
```

- Enviando un mensaje a un objeto (método)

```
objeto.metodo();
```

- Como constructor

```
new MiConstructor();
```

- Indirectamente, a través de `call(...)` y `apply(...)`

```
fn.call({}, "param");
```

# Repaso: Mensajes

- Como constructor

```
new MiConstructor();
```

# Constructores

- Funciones
- Invocación precedida por **new**
- Su contexto es un objeto recién generado
- **return** implícito
- La única manera de manipular prototipos

# Constructores

```
function Constructor(param) {
 // this tiene otro significado!
 this.propiedad = "una propiedad!";
 this.cena = param;
}

var instancia = new Constructor("Pollo asado");
instancia.propiedad; // una propiedad!
instancia.cena; // "Pollo asado"
```

# Constructores

```
function Constructor(param) {
 // this tiene otro significado!
 this.propiedad = "una propiedad!";
 this.cena = param;
}
```

```
var instancia = new Constructor("Pollo asado");
instancia.propiedad; // una propiedad!
instancia.cena; // "Pollo asado"
```

# Constructores

3 pasos:

1. Crear un nuevo objeto
2. Prototipo del objeto = propiedad **prototype** del constructor
3. El nuevo objeto es el contexto del constructor

# Constructores

```
var b = {
 uno: 1,
 dos: 2
};

function A() {
 this.tres = 3;
 this.cuatro = 4;
}

A.prototype = b;

var instancia = new A();
instancia.tres; // 3
instancia.uno; // 1
```

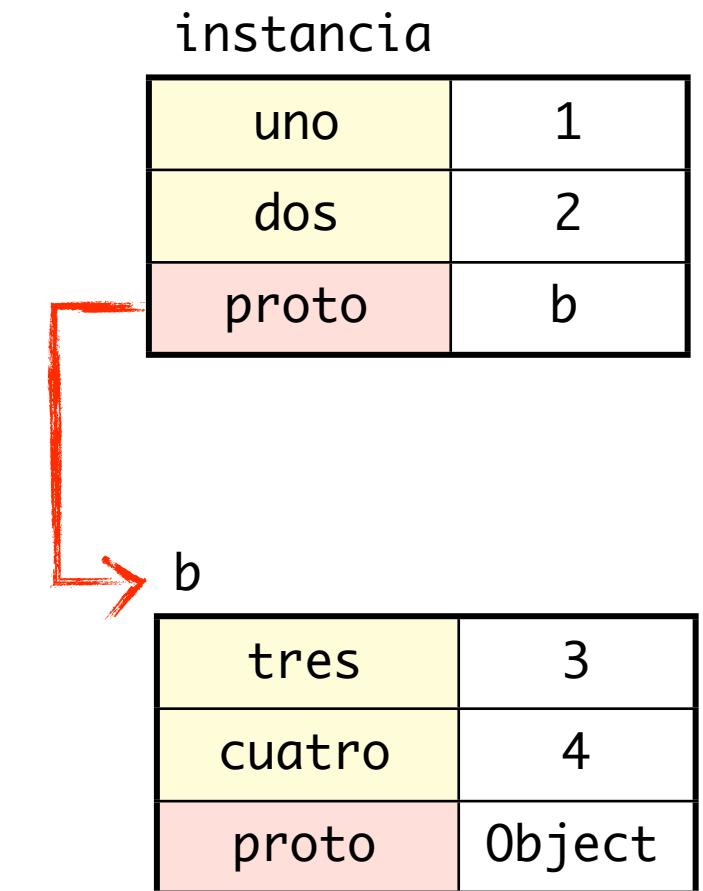
# Constructores

```
var b = {
 uno: 1,
 dos: 2
};

function A() {
 this.tres = 3;
 this.cuatro = 4;
}

A.prototype = b;

var instancia = new A();
instancia.tres; // 3
instancia.uno; // 1
```



# Constructores

## .hasOwnProperty(name)

- Distinguir las propiedades heredadas de las propias
- **true** solo si la propiedad es del objeto

```
instancia.hasOwnProperty("tres"); // true
instancia.hasOwnProperty("uno"); // false
```

# Constructores

¿Qué pasa aquí?

```
var comun = { empresa: "ACME" };

function Empleado(nombre) {
 this.nombre = nombre;
}
Empleado.prototype = comun;

var pepe = new Empleado("Pepe");

pepe.nombre; // "Pepe"
pepe.empresa; // ???
```

# Constructores

¿Qué pasa aquí?

```
var comun = { empresa: "ACME" };

function Empleado(nombre) {
 this.nombre = nombre;
}
Empleado.prototype = comun;

var pepe = new Empleado("Pepe");

comun.empresa = "Googlezon";
var antonio = new Empleado("Antonio");

antonio.empresa; // ???
```

# Constructores

¿Qué pasa aquí?

```
var comun = { empresa: "ACME" };

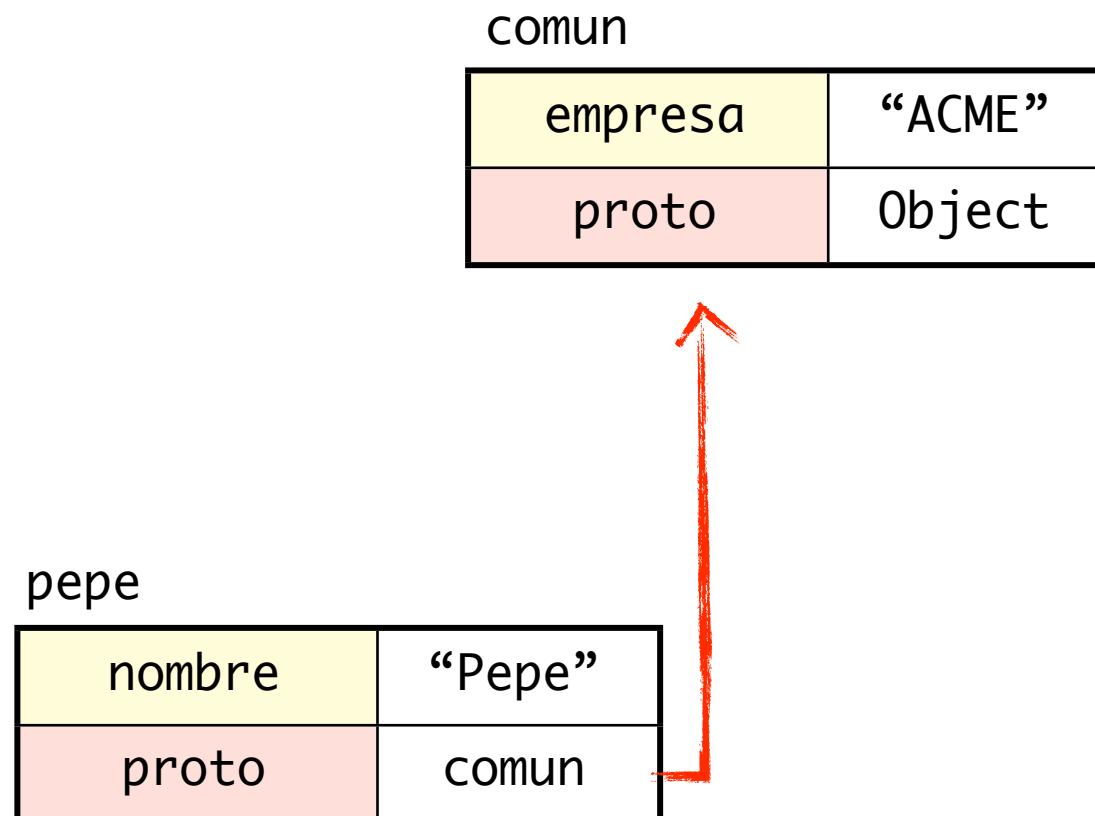
function Empleado(nombre) {
 this.nombre = nombre;
}
Empleado.prototype = comun;

var pepe = new Empleado("Pepe");

comun.empresa = "Googlezon";
var antonio = new Empleado("Antonio");

pepe.empresa; // ???
```

# Constructores



```
var pepe = new Empleado("Pepe");
```

# Constructores

comun

|         |              |
|---------|--------------|
| empresa | “Googlezone” |
| proto   | Object       |

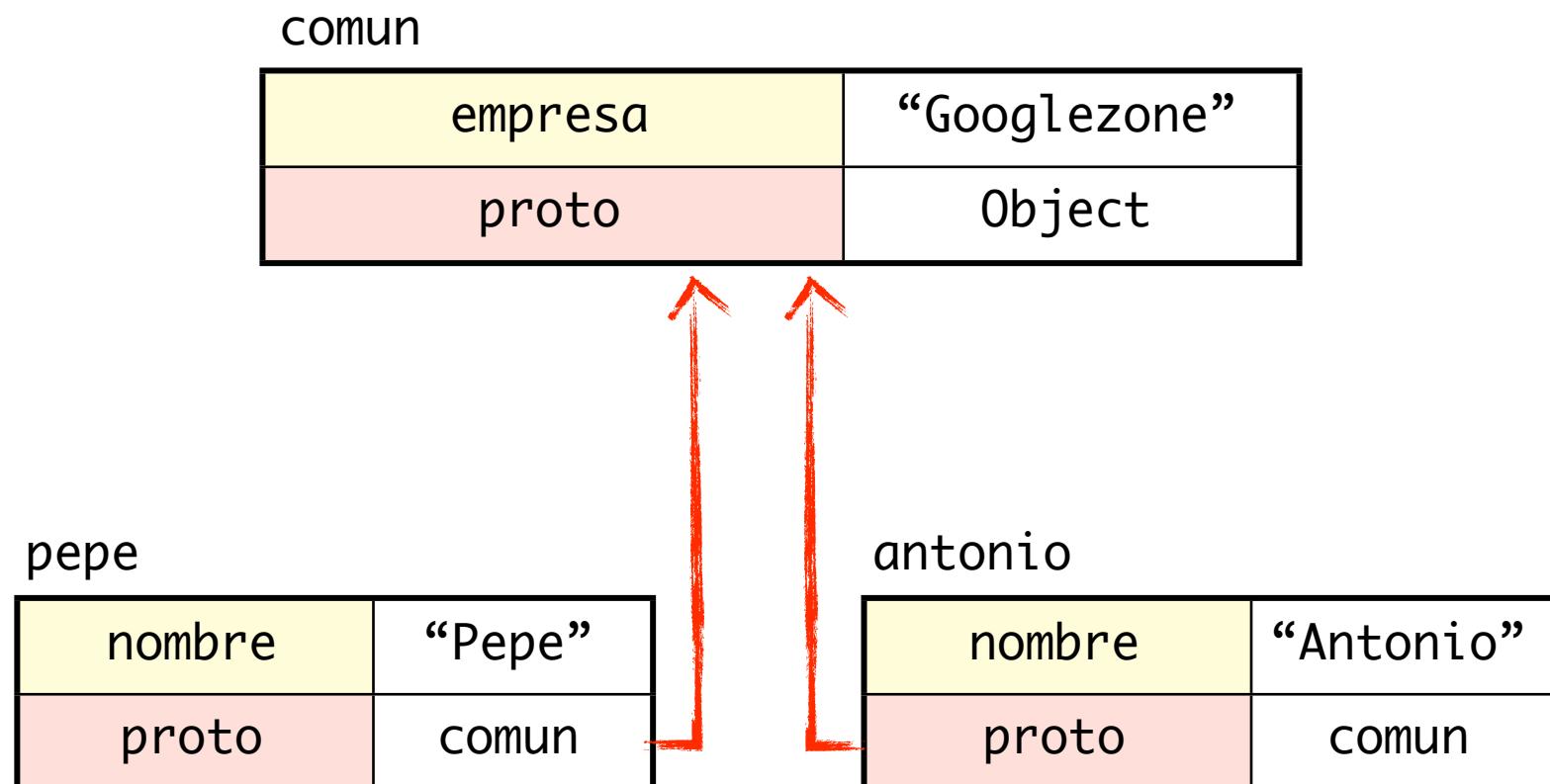


pepe

|        |        |
|--------|--------|
| nombre | “Pepe” |
| proto  | comun  |

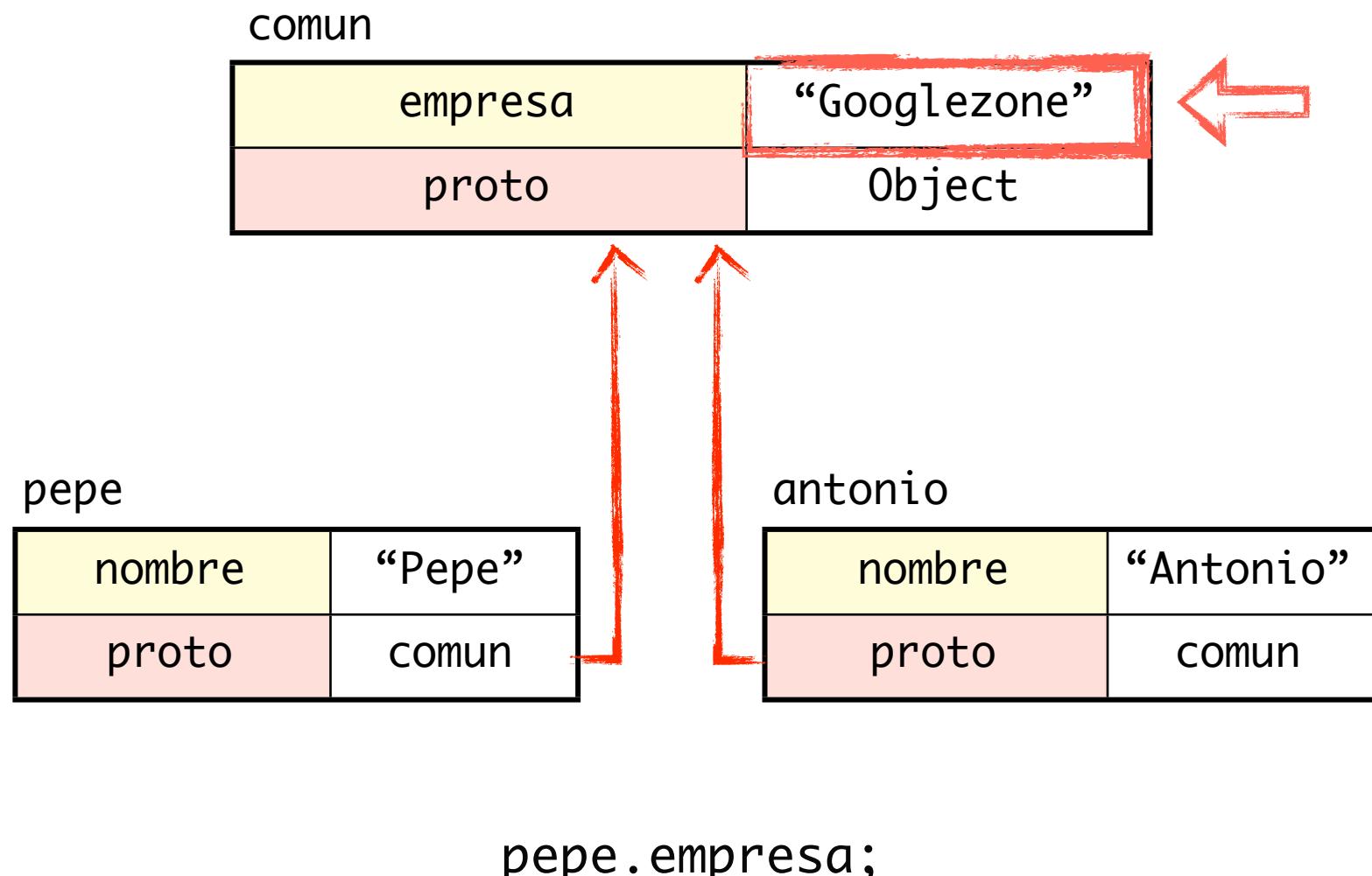
```
comun.empresa = "Googlezon";
```

# Constructores



```
var antonio = new Empleado("Antonio");
```

# Constructores



# Prototipos

Es decir:

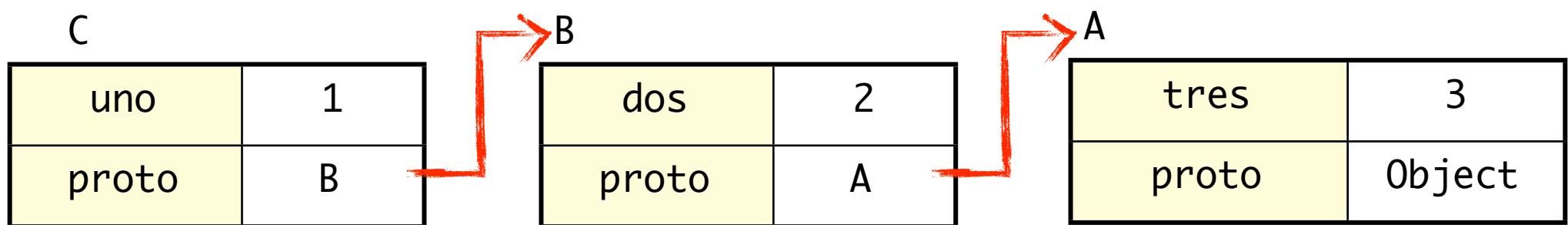
- Las propiedades de los prototipos se comparten!
- Se resuelven dinámicamente
- Modificar un prototipo afecta a todas las instancias anteriores (y futuras)!

# **Intermedio: constructores**

¿Cómo hacer que C herede de B que hereda de A?

# Intermedio: constructores

¿Cómo hacer que C herede de B que hereda de A?



```
var instancia = new C();
instancia.tres; // 3
```

# Intermedio: constructores

```
function C() {
 this.uno = 1;
}

var instancia = new C();
instancia.tres;
```

# Intermedio: constructores

```
var B = {dos: 2};
```

```
function C() {
 this.uno = 1;
}
```

```
C.prototype = B;
```

```
var instancia = new C();
instancia.tres;
```

# Intermedio: constructores

```
var A = {tres: 3};

function B() {
 this.dos = 2;
}
B.prototype = A;

function C() {
 this.uno = 1;
}
C.prototype = B;

var instancia = new C();
instancia.tres;
```

# Intermedio: constructores

```
var A = {tres: 3};

function B() {
 this.dos = 2;
}
B.prototype = A;

function C() {
 this.uno = 1;
}
C.prototype = B;

var instancia = new C();
instancia.dos; // !!!
```

# Intermedio: constructores

```
var A = {tres: 3};

function B() {
 this.dos = 2;
}
B.prototype = A;

function C() {
 this.uno = 1;
}
C.prototype = B;

typeof C.prototype; // ???
C.prototype.dos; // ???
```

# Intermedio: constructores

```
var A = {tres: 3};
```

```
function B() {
 this.dos = 2;
}
B.prototype = A;
```

```
function C() {
 this.uno = 1;
}
C.prototype = new B();
```

```
var instancia = new C();
instancia.tres;
```

# Intermedio: constructores

```
function A() {
 this.tres = 3;
}
```

```
function B() {
 this.dos = 2;
}
B.prototype = new A();
```

```
function C() {
 this.uno = 1;
}
C.prototype = new B();
```

```
var instancia = new C();
instancia.tres;
```

# Cadena de prototipos

La herencia en varios niveles necesita:

- Encadenar prototipos
- El prototipo del “sub constructor” ha de ser siempre `new Padre()`
- Es la única manera de mantener el “padre del padre” en la cadena!

# **Mecanismos de herencia**

# Mecanismos de herencia

- Herencia clásica
- Herencia de prototipos
- Mixins (módulos)
- Extra: herencia funcional

# Herencia clásica

¿Qué significa?

- Clases!
- El tipo de herencia más común en otros lenguajes
- Encapsulado y visibilidad

# Herencia clásica

Vamos a empezar por un constructor:

```
function MiClase() {
 // ???
}

var instancia = new MiClase();
```

# Herencia clásica

Las propiedades (públicas)

```
function MiClase() {
 this.unaPropiedad = "valor";
 this.otraPropiedad = "otro valor";
}

var instancia = new MiClase();
```

# **Herencia clásica**

¿Métodos?

# Herencia clásica

¿Métodos?

```
function MiClase() {
 this.unaPropiedad = "valor";
 this.otraPropiedad = "otro valor";

 this.unMetodo = function() {
 alert(this.unaPropiedad);
 };

 this.otroMetodo = function() {
 alert(this.otraPropiedad);
 };
}

var instancia = new MiClase();
instancia.otroMetodo();
```

# Herencia clásica

Mejor así:

```
function MiClase() {
 this.unaPropiedad = "valor";
 this.otraPropiedad = "otro valor";
}

MiClase.prototype.unMetodo = function() {
 alert(this.unaPropiedad);
};

MiClase.prototype.otroMetodo = function() {
 alert(this.otraPropiedad);
};

var instancia = new MiClase();
instancia.otroMetodo();
```

# Herencia clásica

Solo falta...

```
function Superclase() { /* ... */ }
```

```
function MiClase() {
 this.unaPropiedad = "valor";
 this.otraPropiedad = "otro valor";
}
```

```
MiClase.prototype = new Superclase();
```

```
MiClase.prototype.unMetodo = function() {
 alert(this.unaPropiedad);
};
```

```
MiClase.prototype.otroMetodo = function() {
 alert(this.otraPropiedad);
};
```

# Herencia clásica

¿Cómo se pueden crear métodos “de clase”?

```
MiClase.metodoEstatico("hola!");
```

# Herencia clásica

¿Cómo se pueden crear métodos “de clase”?

```
MiClase.metodoEstatico("hola!");
```

¡Los constructores son objetos!

```
MiClase.metodoEstatico = function(cadena) {
 console.log("metodoEstatico:", cadena);
}
```

# **Herencia clásica: Cuidado!**

Este esquema tiene varios problemas:

# **Herencia clásica: Cuidado!**

Este esquema tiene varios problemas:

- ¡No es fácil invocar al super constructor!

# Herencia clásica: Cuidado!

```
function Animal(especie) {
 this.especie = especie;
}
```

```
Animal.prototype.getEspecie = function() {
 return this.especie;
}
```

```
function Perro(raza) {
 this.raza = raza;
}
Perro.prototype = new Animal();
```

```
Perro.prototype.describir = function() {
 return this.getEspecie() + ", de raza " + this.raza;
}
```

# Herencia clásica: Cuidado!

```
function Animal(especie) {
 this.especie = especie;
}
```

```
Animal.prototype.getEspecie = function() {
 return this.especie;
}
```

```
function Perro(raza) {
 // ???
 this.raza = raza;
}
```

```
Perro.prototype = new Animal();
```

```
Perro.prototype.describir = function() {
 return this.getEspecie() + ", de raza " + this.raza;
}
```

# Herencia clásica: Cuidado!

```
function Animal(especie) {
 this.especie = especie;
}
```

```
Animal.prototype.getEspecie = function() {
 return this.especie;
}
```

```
function Perro(raza) {
 Animal("perro");
 this.raza = raza;
}
```

```
Perro.prototype = new Animal();
```

```
Perro.prototype.describir = function() {
 return this.getEspecie() + ", de raza " + this.raza;
}
```

# Herencia clásica: Cuidado!

```
function Animal(especie) {
 this.especie = especie;
}
```

```
Animal.prototype.getEspecie = function() {
 return this.especie;
}
```

```
function Perro(raza) {
 Animal("perro");
 this.raza = raza;
}
```

```
Perro.prototype = new Animal();
```

```
Perro.prototype.describir = function() {
 return this.getEspecie() + ", de raza " + this.raza;
}
```

**crea glob. especie!**

# Herencia clásica: Cuidado!

```
function Animal(especie) {
 this.especie = especie;
}
```

```
Animal.prototype.getEspecie = function() {
 return this.especie;
}
```

```
function Perro(raza) {
 this = new Animal("perro");
 this.raza = raza;
}
```

```
Perro.prototype = new Animal();
```

```
Perro.prototype.describir = function() {
 return this.getEspecie() + ", de raza " + this.raza;
}
```

# Herencia clásica: Cuidado!

```
function Animal(especie) {
 this.especie = especie;
}
```

```
Animal.prototype.getEspecie = function() {
 return this.especie;
}
```

```
function Perro(raza) {
 this = new Animal("perro");
 this.raza = raza;
}
```

```
Perro.prototype = new Animal();
```

```
Perro.prototype.describir = function() {
 return this.getEspecie() + ", de raza " + this.raza;
}
```

**ERROR!**

# Herencia clásica: Cuidado!

```
function Animal(especie) {
 this.especie = especie;
}
```

```
Animal.prototype.getEspecie = function() {
 return this.especie;
}
```

```
function Perro(raza) {
 Animal.call(this, "perro");
 this.raza = raza;
}
```

```
Perro.prototype = new Animal();
```

```
Perro.prototype.describir = function() {
 return this.getEspecie() + ", de raza " + this.raza;
}
```

# **Herencia clásica: Cuidado!**

Este esquema tiene varios problemas:

- ¡No es fácil invocar al super constructor!
- No es fácil encapsular

# Herencia clásica: Cuidado!

```
function MiClase() {
 this.propPublica = "pública!";
}

MiClase.prototype.metPublico = function() {
 return "público!";
}

var instancia = new MiClase();
instancia.propPublica; // "pública!"
instancia.metPublico();
```

# **Herencia clásica: Cuidado!**

Este esquema tiene varios problemas:

- ¡No es fácil invocar al super constructor!
- No es fácil encapsular...
- ¡Se crea una instancia solo para mantener la cadena de prototipos!

# Herencia clásica: Cuidado!

```
function Superclase() {
 operacionMuyCostosa();
 alert("Oh, no!");
}

function MiClase() {
 // ...
}
MiClase.prototype = new Superclase();
```

# **Herencia clásica?**

¿Qué se puede hacer?

# **Herencia clásica?**

Hay dos enfoques:

# **Herencia clásica?**

Hay dos enfoques:

- El simple

# El “simple”

“Hagamos una funcioncita!”

```
function inherits(subClass, superClass) {
 var F = function() {};
 F.prototype = superClass.prototype;
 subClass.prototype = new F();
 subClass.prototype.constructor = subClass;
 // extra
 subClass.prototype.superclass = superClass;
}
```

# El “simple”

```
function Animal() { }
Animal.prototype.mover = function() {
 console.log("El animal se mueve...");
};

Animal.prototype.comer = function() {
 console.log("¡Ñam!");
};

function Perro(raza) {
 this.superclass.call(this);
}
inherits(Perro, Animal); ←

Perro.prototype.comer = function() {
 console.log("El perro va a por su plato...");
 this.superclass.prototype.comer.call(this);
};

var p = new Perro("terrier");
p.mover();
p.comer();
p instanceof Perro;
```

# El “simple”

```
function Animal() { }
Animal.prototype.mover = function() {
 console.log("El animal se mueve...");
};

Animal.prototype.comer = function() {
 console.log("¡Ñam!");
};

function Perro(raza) {
 this.superclass.call(this); ←
}
inherits(Perro, Animal);

Perro.prototype.comer = function() {
 console.log("El perro va a por su plato...");
 this.superclass.prototype.comer.call(this);
};

var p = new Perro("terrier");
p.mover();
p.comer();
p instanceof Perro;
```

**OMG!**

# El “simple”

- Ventajas
  - Muy simple de implementar
  - Muy ligero
  - No añade demasiado ruido
- Inconvenientes
  - No soluciona mucho...
  - No se “heredan” los métodos/propiedades de clase
  - Sigue sin ser cómodo de usar

# El “simple”

Caso práctico: CoffeeScript

```
var __hasProp = {}.hasOwnProperty,
 __extends = function (child, parent) {
 for (var key in parent) {
 if (__hasProp.call(parent, key))
 child[key] = parent[key];
 }
 function ctor() {
 this.constructor = child;
 }
 ctor.prototype = parent.prototype;
 child.prototype = new ctor();
 child.__super__ = parent.prototype;
 return child;
 };
}
```

# El “simple”

Caso práctico: CoffeeScript

```
var __hasProp = {}.hasOwnProperty,
 __extends = function (child, parent) {
 for (var key in parent) {
 if (__hasProp.call(parent, key))
 child[key] = parent[key];
 }
 function ctor() {
 this.constructor = child;
 }
 ctor.prototype = parent.prototype;
 child.prototype = new ctor();
 child.__super__ = parent.prototype;
 return child;
 };
```

# El “simple”

```
var MiClase = (function(_super) {
 → __extends(MiClase, _super);

 → function MiClase() {
 MiClase.__super__.constructor.apply(this, arguments);
 this.miPropiedad = 1;
 }

 MiClase.prototype.miMetodo = function() {
 return MiClase.__super__.miMetodo.call(this, "hola");
 };

 return MiClase;
})(Superclase);
```

# **Herencia clásica?**

Hay dos enfoques:

- El simple
- El cómodo

# El cómodo

Más complejo, pero merece la pena

```
var Persona = Class.extend({
 init: function(nombre) {
 console.log("Bienvenido,", nombre);
 }
});
```

```
var Ninja = Persona.extend({
 init: function(){
 this._super("ninja");
 }
 esgrimirEspada: function(){
 console.log("En guardia!");
 }
});
```

# El cómodo

Más complejo, pero merece la pena

```
→ var Persona = Class.extend({
 init: function(nombre) {
 console.log("Bienvenido,", nombre);
 }
});
```

```
→ var Ninja = Persona.extend({
 init: function(){
 this._super("ninja");
 }
 esgrimirEspada: function(){
 console.log("En guardia!");
 }
});
```

# Intermedio: klass.js

¡Rellena los huecos!

```
var Class = function(){};

Class.extend = function(prop) {
 var _super = this.prototype;

 // ...

 return Klass;
};
```

# Intermedio: klass.js

Está muy bien pero...

- No hay métodos de clase (y no se heredan!)
- Todo sigue siendo público
- ¡Es solo una primera versión!

# **El cómodo**

¿Cuándo usar este método?

- ¡Siempre que sea posible!

Contras:

- La implementación es más compleja...
- Hay que incluir la librería externa
- No es el enfoque “tradicional”

# **Herencia de prototipos**

Vamos a cambiar de marcha...

# **Herencia de prototipos**

Vamos a cambiar de marcha...

¡Ahora, sin clases!

# **Herencia de prototipos**

Vamos a cambiar de marcha...

¡Ahora, sin clases!

¿¿Cómo puede haber POO sin clases??

# **Herencia de prototipos**

Herencia clásica: categorías

- Definir “Persona”
- crear instancias de persona según la definición
- Para hacer más concreto, ¡redefine!

# **Herencia de prototipos**

Herencia clásica: categorías

- Definir “Persona”
- crear instancias de persona según la definición
- Para hacer más concreto, ¡redefine!

Herencia de prototipos: ejemplos

- “Como ese de ahí, pero más alto”
- Cualquier objeto concreto puede servir de ejemplo
- Para hacer más concreto, ¡cambia lo que quieras!

# Herencia de prototipos

```
var benito = {
 nombre: "Benito",
 edad: 36,
 profesión: "jardinero",
 saludar: function() {
 alert("Buen día!");
 },
 envejecer: function() {
 this.edad += 1;
 }
};
```

# Herencia de prototipos

```
var benito = {
 nombre: "Benito",
 edad: 36,
 profesión: "jardinero",
 saludar: function() {
 alert("Buen día!");
 },
 envejecer: function() {
 this.edad += 1;
 }
};

var gonzalo = clone(benito);
gonzalo.nombre = "Gonzalo";
gonzalo.profesion = "carpintero";
gonzalo.saludar(); // Buen día!
```

# Herencia de prototipos

```
var benito = {
 nombre: "Benito",
 edad: 36,
 profesión: "jardinero",
 saludar: function() {
 alert("Buen día!");
 },
 envejecer: function() {
 this.edad += 1;
 }
};
```



```
var gonzalo = clone(benito);
gonzalo.nombre = "Gonzalo";
gonzalo.profesion = "carpintero";
gonzalo.saludar(); // Buen día!
```

# Herencia de prototipos

También se puede generalizar

```
var Animal = {
 vivo: true,
 comer: function() {
 console.log("Ñam, ñam");
 }
};
```

```
var Perro = clone(Animal);
Perro.especie = "perro";
```

```
var Dogo = clone(Perro);
Dogo.raza = "dogo";
```

```
var toby = clone(Dogo);
toby.nombre = "Toby";
```

# Herencia de prototipos

¡Así de simple!

```
function clone(obj) {
 function F(){}
 F.prototype = obj;
 return new F();
}
```

# Herencia de prototipos

Herencia clásica vs. de prototipos

- Clásica

- ✓ MUCHO más extendida y bien comprendida
- ✓ Mayor catálogo de mecanismos de abstracción

- Prototipos

- ✓ Uso mucho más eficiente de la memoria
- ✓ La “auténtica” herencia en JS
- ✓ Muy simple

# Herencia de prototipos

Peero...

- Clásica
  - ◉ Solo se puede emular. Necesario entender los prototipos.
  - ◉ A contrapelo
- Prototipos
  - ◉ Bastante limitada
  - ◉ Lenta con cadenas de prototipos largas!

# **Herencia de prototipos**

¿Cuál uso?

# **Herencia de prototipos**

¿Cuál uso?

¡Las dos!

# Intermedio: prototipos

¿Qué significa?

objeto.propiedad;

# Intermedio: prototipos

¿Qué significa?

objeto.propiedad;

“Accede a la propiedad **propiedad** del objeto **objeto**”

# Intermedio: prototipos

¿Qué significa?

`objeto.propiedad;`

“Accede a la propiedad **propiedad** del objeto **objeto**. Si no la encuentras, sigue buscando por la cadena de prototipos.”

# Intermedio: prototipos

¿Qué significa?

```
objeto.propiedad = 1;
```

# Intermedio: prototipos

¿Qué significa?

```
objeto.propiedad = 1;
```

“Guarda el valor 1 en la propiedad **propiedad** del objeto **objeto**.”

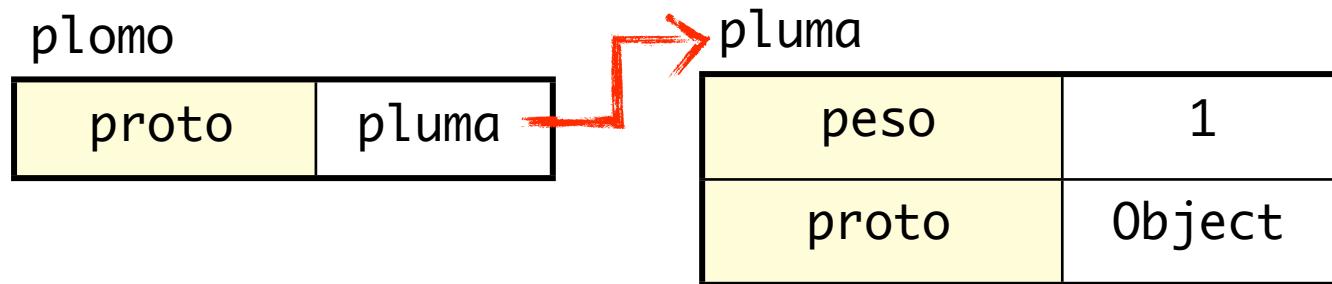
# Intermedio: prototipos

¿Qué significa?

```
objeto.propiedad = 1;
```

“Guarda el valor 1 en la propiedad **propiedad** del objeto **objeto**. Si no la encuentras, créala!”

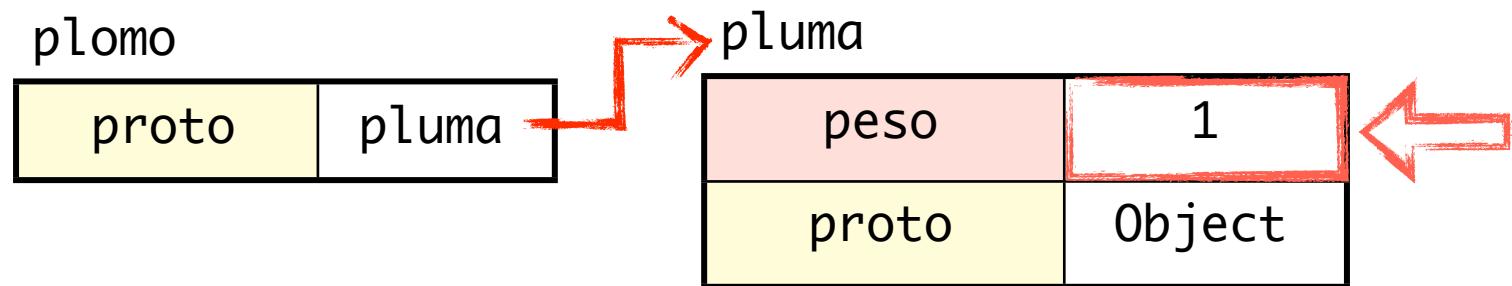
# Intermedio: prototipos



```
var pluma = {
 peso: 1
};
```

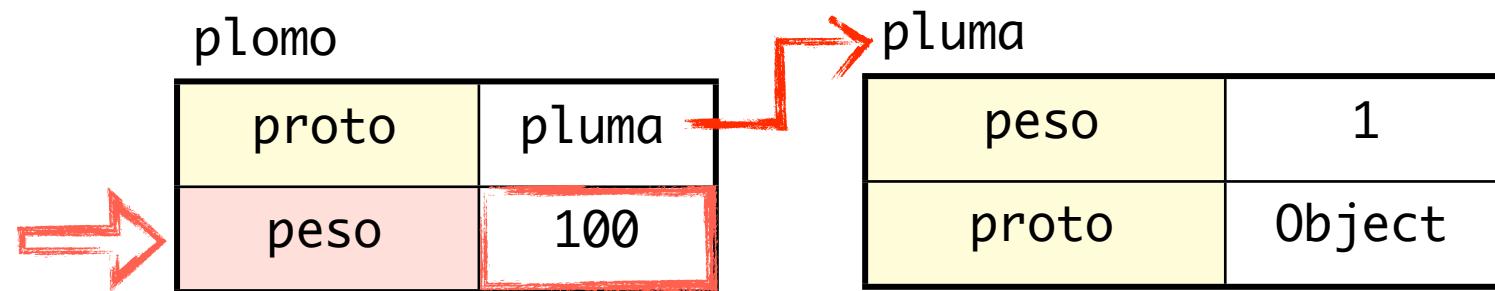
```
var plomo = clone(pluma);
```

# Intermedio: prototipos



```
plomo.peso; // 1
```

# Intermedio: prototipos



```
plomo.peso = 100;
```

# Intermedio: prototipos

Es decir:

- Hay asimetría entre escritura y lectura!
- Lectura: busca en la cadena
- Escritura: crea una nueva propiedad
- Es el comportamiento natural
- Uso eficiente de la memoria: solo se crean los valores diferentes.

# Intermedio: prototipos

¿Qué sucede?

```
var Lista = {
 elementos: []
};
```

```
var laCompra = clone(Lista);
```

```
laCompra.elementos.push("Leche");
laCompra.elementos.push("Huevos");
```

```
var ToDo = clone(Lista);
ToDo.elementos.push("Contestar emails");
ToDo.elementos.push("Subir a producción");
```

```
ToDo.elementos;
```

# prototipos

## Object.create(proto)

- Igual que `clone`
- Nativo en algunos navegadores

```
var pluma = {
 peso: 1,
 enStock: true,
};
```

```
var plomo = Object.create(pluma);
plomo.peso = 100;
```

# **¡Se acabaron los prototipos!**

# Clausuras

Sólo una idea importante más: ámbitos

# Clausuras

- Una idea sencilla, pero difícil de explicar
- Están por todas partes
- Consecuencia natural del lenguaje
- ¡Muy útiles!

# Clausuras

```
function clausurador() {
 var a = 1;
 return function() {
 return a;
 };
}
```

# Clausuras

```
function clausurador() {
 var a = 1;
 return function() {
 return a;
 };
}

var fn = clausurador();
typeof fn;
```

# Clausuras

```
function clausurador() {
 var a = 1;
 return function() {
 return a;
 };
}

var fn = clausurador();
fn(); // ???
```

# Clausuras

```
function clausurador() {
 var a = 1;
 return function() {
 return a;
 };
}
```

```
fn = function() {
 return a;
}
```

```
var fn = clausurador();
fn();
```

# Clausuras

```
function clausurador() {
 var a = 1;
 return function() {
 return a;
 };
}
```

```
var fn = clausurador();
fn();
```

```
fn = function() {
 return a;
}
```

# Clausuras

```
→ function clausurador() {
 var a = 1;
 return function() {
 return a;
 };
}
```

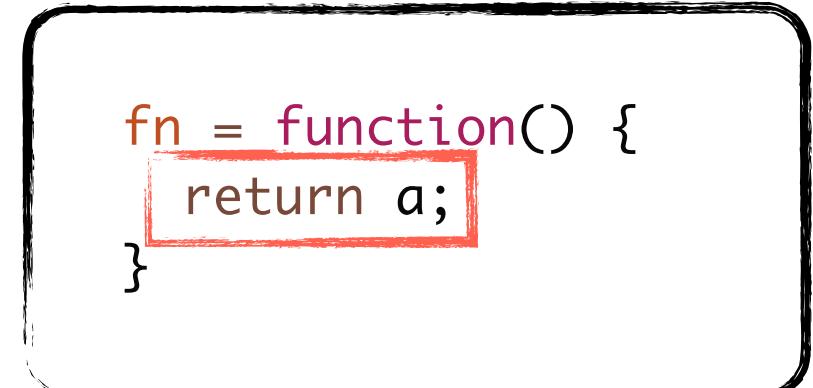
```
var fn = clausurador();
fn();
```

```
fn = function() {
 return a;
}
```

# Clausuras

```
→ function clausurador() {
 var a = 1;
 return function() {
 return a;
 };
}
```

```
var fn = clausurador();
fn();
```



# Clausuras

Otro caso:

```
function makeContador() {
 var i = 0;
 return function() {
 return i++;
 }
}
```

```
var contador1 = makeContador();
contador1(); // ???
contador1(); // ???
```

```
var contador2 = makeContador();
contador2(); // ???
```

# Clausuras

Otro caso:

```
function makeContador() {
 var i = 0;
 return function() {
 return i++;
 }
}
```

```
var contador1 = makeContador();
contador1(); // 0
contador1(); // 1
```

```
var contador2 = makeContador();
contador2(); // 0
```

# Clausuras

```
function makeContador() {
 var i = 0;
 return function() {
 return i++;
 }
}
```

```
contador1 = function() {
 return i++;
}
```

```
var contador1 = makeContador();
contador1(); // 0
contador1(); // 1
```

```
var contador2 = makeContador();
contador2(); // 0
```

i = 0;

# Clausuras

```
function makeContador() {
 var i = 0;
 return function() {
 return i++;
 }
}
```

```
contador1 = function() {
 return i++;
}
```

```
var contador1 = makeContador();
contador1(); // 0
contador1(); // 1
```

```
var contador2 = makeContador();
contador2(); // 0
```

i = 1;

# Clausuras

```
function makeContador() {
 var i = 0;
 return function() {
 return i++;
 }
}

var contador1 = makeContador();
contador1(); // 0
contador1(); // 1

var contador2 = makeContador();
contador2(); // 0
```

```
contador1 = function() {
 return i++;
}
```

i = 1;

```
contador2 = function() {
 return i++;
}
```

i = 0;

# Clausuras

```
function interesante() {
 var algo = 0;
 return {
 get: function() {
 return algo;
 },
 set: function(valor) {
 return algo = valor;
 }
 };
}

var obj = interesante();
obj.get(); // ???
obj.set("hola!");
obj.get(); // ???
```

# Clausuras

- `bind`: fija una función a un contexto

```
function bind(ctx, fn) {
 return function() {
 return fn.apply(ctx, arguments);
 }
}
```

# Clausuras

- **curry**: aplicación parcial de una función

```
function curry(fn) {
 var slice = Array.prototype.slice,
 args = slice.call(arguments, 1);
 return function() {
 var newargs = slice.call(arguments);
 return fn.apply(this, [args].concat(newargs));
 };
}
```

# Clausuras

Es decir:

- Las clausuras son función + entorno
- Asocian datos a funciones
- No se puede acceder directamente a las variables clausuradas desde el exterior de la función
- Duración indefinida
- ¡Son muy útiles!

# Intermedio: herencia funcional

```
function PseudoConstructor() {
 var self = {};
 self.propiedad = "valor";
 return self;
}

var pseudoInstancia = PseudoConstructor();
pseudoInstancia.propiedad; // "valor"
```

# Intermedio: herencia funcional

```
function PseudoConstructor() {
 var self = {};
 self.propiedad = "valor";
 self.metodo = function(nombre) {
 return "No te duermas, " + nombre + "!";
 };
 return self;
}

var pseudoInstancia = PseudoConstructor();
pseudoInstancia.metodo("Abraham");
```



# Intermedio: herencia funcional

```
function PseudoConstructor() {
 var self = {};
 self.propiedad = "valor";
 self.metodo = function(nombre) {
 return "No te duermas, " + nombre + "!";
 };
 return self;
}

var pseudoInstancia = PseudoConstructor();
pseudoInstancia.metodo("Abraham");
```

# Intermedio: herencia funcional



```
function PseudoConstructor() {
 var self = {},
 propiedad = "privada!";
 var metodoPrivado = function(s) {
 return s.toUpperCase();
 }
 self.metodo = function(nombre) {
 var mayus = metodoPrivado(nombre);
 return "No te duermas, " + mayus + "!";
 };
 return self;
}

var pseudoInstancia = PseudoConstructor();
pseudoInstancia.metodo("Abraham");
```

# Intermedio: herencia funcional

1. Crear un pseudoconstructor
2. Definir una variable `self` con un objeto vacío
3. Añadir las propiedades/métodos públicos a `self`
4. Devolver `self`

# **Intermedio: herencia funcional**

¿Y para heredar?

# Intermedio: herencia funcional

¿Y para heredar?

```
function A() {
 var self = {};
 self.uno = 1;
 return self;
}
```



```
function B() {
 var self = A();
 self.dos = 2;
 return self;
}
```

```
var b = B();
b.uno; // 1
```

# **Intermedio: herencia funcional**

¿Cómo llamar al supermétodo?

# Intermedio: herencia funcional

```
function A() {
 var self = {};
 self.metodo = function() {
 console.log("A");
 }
 return self;
}

function B() {
 var self = A();
 var superMetodo = self.metodo;
 self.metodo = function() {
 superMetodo();
 console.log("B");
 }
 return self;
}
```



# Intermedio: herencia funcional

“Herencia funcional”:

- ✓ Explotar clausuras y objetos en linea
- ✓ Extremadamente simple e intuitivo
- ✓ Mejor encapsulado público/privado
- ✓ Poco ruido sintáctico
- ✓ No hacen falta helpers ni librerías

# Intermedio: herencia funcional

“Herencia funcional”:

- ✓ Explotar clausuras y objetos en linea
- ✓ Extremadamente simple e intuitivo
- ✓ Mejor encapsulado público/privado
- ✓ Poco ruido sintáctico
- ✓ No hacen falta helpers ni librerías
- Un poco... ¿cutre?
- No es la manera más popular
- ¡Peor uso de la memoria!