Fundamentos

Declaración de variables

Declaración de variables

- ES6 introduce la sentencia **let**
  - cumple la misma misión que **var**
  - se comporta ligeramente diferente

```
function myFunc() {
  console.log('valor: ', x)
  var x = 12
  console.log('valor: ', x)
}

myFunc()
```

```
function myFunc() {
  var x
  console.log('valor: ', x)
  x = 12
  console.log('valor: ', x)
}

myFunc()
```

```
function myFunc() {
  console.log('valor: ', x)
  let x = 12
  console.log('valor: ', x)
}

myFunc()
```

```
function myLoop() {
  for (var i=0; i <= 10; i++) {
    // no-op
  }
  return i
}
```

```
function myLetLoop() {
  for (let i=0; i <= 10; i++) {
    // no-op
  }
  return i
}
```

Ejercicio

- Encuentra y arregla el bug

```
function createFns() {
  let fns = []
  for (var i = 0; i< 10; i++) {
    fns.push(function() { console.log(i) })
  }
  return fns
}
```

Ejercicio

- Encuentra y arregla el bug

```
function randomNumber(n) {
  if (Math.random() > .5) {
    let base = 1
  } else {
    let base = -1
  }
  return base * n * Math.random()
}
```

```
function myFunc() {
  let a = 1
  let b = 0
  for (let i=4; i--;) {
    let b = a + 1
  }
  return b
}
```

```
function myFunc() {
  let a = 1
  for (let i=4; i--;) {
    let b = a + 1
  }
  return b
}
```

```
function myFunc() {
  let a = 1
  for (let i=4; i--;) {
    let a = a + 1
  }
  return a
}
```

```
function myFunc() {
  let a = 1
  for (let i=4; i--;) {
    let a = i + 1
  }
  return a
}
```

```
const uno = 1
```

```
const uno = 1;
uno = 2; // ERROR! Assignment to constant variabe
```

redradix

Tipos de Datos Primitivos

Tipos de Datos

- Javascript ofrece **6** tipos de datos primitivos

Tipos de Datos

- Javascript ofrece **6** tipos de datos primitivos
  - Boolean
  - Number
  - String
  - Symbol
  - Null
  - Undefined

Tipos de Datos

- Operador **`typeof`**
  - Informa del tipo de un dato dado

Tipos de Datos

```
typeof 42
```

Tipos de Datos

```
typeof "42"
```

Tipos de Datos

```
typeof undefined
```

Tipos de Datos

```
typeof null
```

String templates

```
const dinamico = 'contenido interpolado';
const final = `Esto es literal, esto es ${dinamico}`;
console.log(final);
```

```
const dinamico = 'contenido interpolado';
const final = `Esto es literal, esto es ${dinamico}`;
console.log(final);
```

```
const dinamico = 'contenido interpolado';
const final = `Esto es literal, esto es ${dinamico}`;
console.log(final);
```

Ejercicio

- Utiliza **string templates** para...
  - crear un programa que muestra la hora (*HH:MM:SS*) por la consola cada segundo

Ejercicio

- Utiliza **string templates** para...
  - crear una función que liste los elementos de un array añadiendo una “y” al final
  - ej: `[1, 2, 3]` => `“1, 2 y 3”`

```
const usuario = {
  nombre: 'Elias',
  apellido: 'Alonso'
}

console.log(`Bienvenido, ${usuario}`)
```

Ejercicio

- ¿Qué puedo **añadir** al objeto **usuario** para que se muestre correctamente al ser interpolado?
    - *pista: ¿cómo convierte javascript un valor a string?*

Symbols

Symbols

- Primer tipo de datos nuevo desde 1997
- Función muy especializada
- Similar a los símbolos de Ruby o Lisp

Symbols

- Diferentes al resto de tipos primitivos de datos
  - No tienen **representación literal**
  - Cada símbolo tiene un **valor único** e irrepetible
  - **Inmutables**
  - **No** se convierten a String automáticamente

Symbols

- No tienen **representación literal**
  - No hay sintaxis para representar su valor
  - Creación mediante función constructora
  - No se puede mostrar su valor por consola

Symbols

```
const a = Symbol();

console.log(a); // Symbol()
```

Symbols

```
const a = Symbol('symbol a');

console.log(a); // Symbol(symbol a)
```

Symbols

- Cada símbolo tiene un **valor único** e irrepetible
  - Todos los símbolos **son diferentes** entre sí

Symbols

```
const a = Symbol();
const b = Symbol();

a === b;
```

Symbols

```
const a = Symbol();
const b = Symbol();

a === b; // false
```

Symbols

```
const a1 = Symbol('a');
const a2 = Symbol('a');

a1 === a2;
```

Symbols

```
const a1 = Symbol('a');
const a2 = Symbol('a');

a1 === a2; // false
```

Symbols

- **No** se convierten a String automáticamente
  - Los demás tipos sí se convierten automáticamente

Symbols

```
const a = Symbol('a');
const str = a + '!';
```

Symbols

```
TypeError: Cannot convert a Symbol value to a string
```

Symbols

`Symbol([description])`

- Crea un **símbolo nuevo con cada invocación**
- Puede recibir, opcionalmente, una *descripción*

Symbols

El estándar especifica algunos símbolos predefinidos

- `Symbol.iterator`
- `Symbol.hasInstance`
- `Symbol.match`

Symbols

Los símbolos se pueden utilizar como

nombres de propiedades

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

console.log(obj[p]); // 'value'
```

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

console.log(obj[p]); // 'value'
```

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

p = null;
```

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

console.log(Object.keys(obj)); // []
```

Symbols

Los símbolos sirven para...

- Crear propiedades
- Sin la referencia al símbolo son inaccesibles

Symbols

Aplicaciones:

- Almacenar **metadata**
- Almacenar info “privada” en **objetos externos**
- Configuraciones y propiedades especiales

Tipos de Datos Compuestos

Tipos de Datos

- Javascript ofrece **1** tipo de dato compuesto

Tipos de Datos

- Javascript ofrece **1** tipo de dato compuesto
  - **Object**

Tipos de Datos

*¿Y los **arrays**?*

Tipos de Datos

```
typeof [1, 2]
```

Tipos de Datos

*¿Y las **funciones**?*

Tipos de Datos

```
typeof console.log
```

Tipos de Datos

“Functions are regular objects with the additional capability of being callable.”

Fuente: MDN

Object

Object

- Un conjunto dinámico de propiedades
  - nombre: `string` o `symbol`
  - valor: cualquier valor
- Puede heredar propiedades de otro objeto
- Manejado por referencia

Object

```
const obj = {};

const obj2 = { prop: 1 };

const obj3 = { ['a' + 'b']: 1 };
```

Object

```
const k = 'a';
const obj1 = { [k]: 1 };
const obj2 = { [k]: 1 };

obj1 === obj2; // ???
```

Object

```
const k = 'a';
const obj1 = { [k]: 1 };
const obj3 = obj1;

obj3.b = 2;

obj3 === obj1; // ???
```

```
const k = 'a'
const obj1 = { [k]: 1 }
const obj3 = obj1

obj3.b = 2

console.log(obj1.b)
```

Object.assign

Object

- **`Object.assign`**
  - Nos permite “fusionar” objetos
  - Asignado las propiedades de un objeto a otro
  - De derecha a izquierda

```
const a = { a: 1 }
const b = { b: 2 }

Object.assign(a, b)

console.log(a)
console.log(b)
```

```
const a = { a: 1 }
const b = { b: 2 }
const c = { c: 3 }

Object.assign(a, b, c)
console.log(b)
```

```
const a = { a: 1 }
const b = { b: 2 }
const c = { c: 3 }
const x = Object.assign(a, b, c)

console.log(x) // { a: 1, b: 2, c: 3 };
```

```
const a = { a: 1 }
const b = { b: 2 }
const c = { c: 3 }

const x = Object.assign(a, b, c)

x === a // ???
```

Ejercicio

- ¿Cómo podemos fusionar **a**, **b** y **c** sin modificar **ninguno** de los tres?

Ejercicio

- Escribe una función **clone** que cree una **copia** del objeto que recibe como primer parámetro.

```
const u1 = { username: 'root', password: 'iamgod' }
const u2 = { username: 'luser', password: '12345' }
const users = { u1: u1, u2: u2 }

const usersCopy = clone(users);
usersCopy.u3 = { username: 'admin', password: 'aDS00Dkxx098Sd' }

console.log(users.u3) // ???

usersCopy.u1.username = 'p0wnd'

console.log(users.u1.username) // ???

users.u1 === usersCopy.u1 // ???
```

Ejercicio

- Modifica **clone** para que prevenga el hack del ejemplo anterior

```
const u1 = { a: { b: { c: 1 } } }
const u2 = { a: { b: { d: 2 } } }

const x = Object.assign({}, u1, u2)
console.log(x.a.b) // ???
```

Ejercicio

- Escribe **merge**, la versión recursiva de **Object.assign**

```
const u1 = { a: { b: { c: 1 } } }
const u2 = { a: { b: { d: 2 } } }

const x = merge({}, u1, u2)
console.log(x.a.b) // { c: 1, d: 2 }
```

```
function merge(base, ...args) {
  Object.assign(base, ...args)
  for (let [key, value] of Object.entries(base))
    if (value instanceof Object)
      base[key] = merge(value, ...args.map(function(arg) {
        return (arg[key] || {})
      }))
  return base
}
```

```
const u1 = { a: { b: { c: 1 } }, b: 3, c: 4 }
const u2 = { a: { b: { d: 2 } }, b: 2 }
const u3 = { x: 3, a: { c: 'hey' } }

const x = merge(u1, u2, u3)
console.log(x)
console.log(u1)
console.log(u2)
```

```
const config = {
  server: {
    hostname: 'myapp.domain.com',
    port: 443,
    protocol: 'https'
  },
  database: {
    host: '192.169.1.2',
    port: 33299
  }
}

const testConfig = merge(config, {
  server: { hostname: 'localhost' },
  database: { host: 'localhost' }
})
```

```
const x = [{ a: 1 }, [{ b: 2 }]]
const y = [{ b: 2 }, [], { c: 'hi' }]

console.log(merge(x, y))
```

Object.defineProperty

Object.defineProperty

- **Object.defineProperty**
  - **configurar** las propiedades de un objeto
    - modificar su **valor**
    - controlar si es o no es **enumerable**
    - controlar si es **de solo lectura**
    - controlar si se puede **volver a configurar**

```
const obj = {}
Object.defineProperty(obj, 'a', {
  value: 1
})

console.log(obj.a) // 1
```

```
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
})

console.log(obj.b) // 2
console.log(obj.c) // 3
```

```
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
})

console.log(obj) // ????
```

```
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
})

console.log(Object.keys(obj)) // ????
```

```
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2, enumerable: true },
  c: { value: 3, enumerable: true }
})

console.log(obj)
```

```
const obj = {}

Object.defineProperty(obj, 'a', { value: 1 })

Object.defineProperty(obj, 'a', {
  value: 2,
  enumerable: true
})
```

```
TypeError: Cannot redefine property: a
```

```
const obj = {}

Object.defineProperty(obj, 'a', {
  value: 1,
  configurable: true
})

Object.defineProperty(obj, 'a', {
  value: 2,
  enumerable: true
})
```

Object

- Descriptor de propiedad:
  - `value` (*undefined*)
  - `enumerable` (*false*)
  - `configurable` (*false*)
  - `writable` (*false*)

getters y setters

Object

- El descriptor de propiedad también puede especificar
  - `get`
  - `set`

Object

```
const obj = {};
Object.defineProperty(obj, 'random', {
  get: function() {
    console.log('Tirando dados...');
    return Math.floor(Math.random() * 100);
  }
});
console.log(obj.random); // Tirando dados... 27
console.log(obj.random); // Tirando dados... 18
```

Object

```
const obj = {};

Object.defineProperty(obj, 'a', {
  get: function() {
    return this.a * 2;
  }
});

obj.a = 2;
console.log(obj.a); // ???
```

Object

```
const temp = { celsius: 0 };

Object.defineProperty(temp, 'fahrenheit', {
  set: function(value) {
    this.celsius = (value - 32) * 5/9;
  },
  get: function() {
    return this.celsius * 9/5 + 32;
  }
});
```

Object

```
temp.fahrenheit = 10;
console.log(temp.celsius); // -12.22

temp.celsius = 30;
console.log(temp.fahrenheit); // 86
```

Object

```
const obj = {};
obj.fahrenheit = temp.fahrenheit;

obj.celsius = -12.22;
console.log(obj.fahrenheit); // ???
```

Ejercicio

- Escribe **withAccessCount**
  - una **funcion**
  - que **recibe** un **objeto** y **un nombre de propiedad**
  - cuenta las veces que **se accede** a esa propiedad

```
const obj = { p: 1 }
withAccessCount(obj, 'p')

obj.p = 12
console.log(obj.p)
console.log(obj.p)

console.log(obj.getAccessCount('p')) // 2
```

```
const obj = { p: 1, j: 2 }
withAccessCount(obj, 'p')
withAccessCount(obj, 'j')

console.log(obj.p)
console.log(obj.p)
console.log(obj.j)

console.log('->', obj.getAccessCount('p')) // 2
console.log('->', obj.getAccessCount('j')) // 1
```

Object

```
const obj = {
  get prop() {
    return this._value
  },
  set prop(value) {
    this._value = value * 2
  }
}
```

Ejercicio

- Añade una propiedad **average** a un **array**
  - que devuelva **la media de los valores** del array

Ejercicio

- Escribe un **setter**
  - que guarde todos los valores que se asignan a la propiedad en un array
- Escribe un **getter**
  - que devuelva siempre el último valor del array
- Escribe un método **undo**
  - que restaure el valor anterior de la propiedad

Sellar objetos

redradix

Object

`Object.seal(obj)`

- Finaliza la configuración de propiedades
  - No se pueden añadir nuevas propiedades
  - No se pueden eliminar propiedades
  - Se pueden modificar las propiedades existentes

Object

```
const obj = { a: 1, b: 2, c: 3 };

Object.seal(obj);

obj.c = 0;
obj.d = 4;
console.log(obj); // { a: 1, b: 2, c: 0 }

delete obj.a;
console.log(obj);  // { a: 1, b: 2, c: 0 }
```

Object

`Object.freeze(obj)`

- Inmutabiliza el objeto
  - No se pueden añadir nuevas propiedades
  - No se pueden eliminar propiedades
  - No pueden modificar las propiedades existentes

Object

```
const obj = { a: 1, b: 2, c: 3 };

Object.freeze(obj);

obj.c = 0;
obj.d = 4;
delete obj.a;
console.log(obj); // { a: 1, b: 2, c: 3 }
```

Object.create

Object

`Object.create(proto, properties)`

- Genera un nuevo objeto
  - ***proto***: prototipo del objeto
  - ***properties***: descriptores de propiedades

Object

```
const obj = { a: 1, b: 2 };
console.log(obj); // { a: 1, b: 2 }
console.log(obj.toString); // ???
```

Object

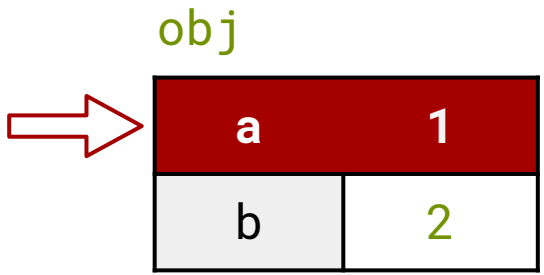
```
const obj = { a: 1, b: 2 };
```

obj

a	1
b	2

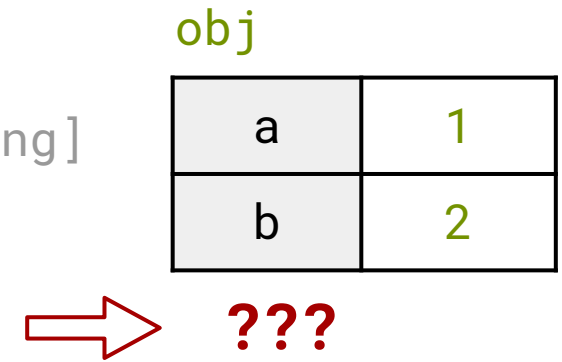
Object

```
obj.a // 1
```



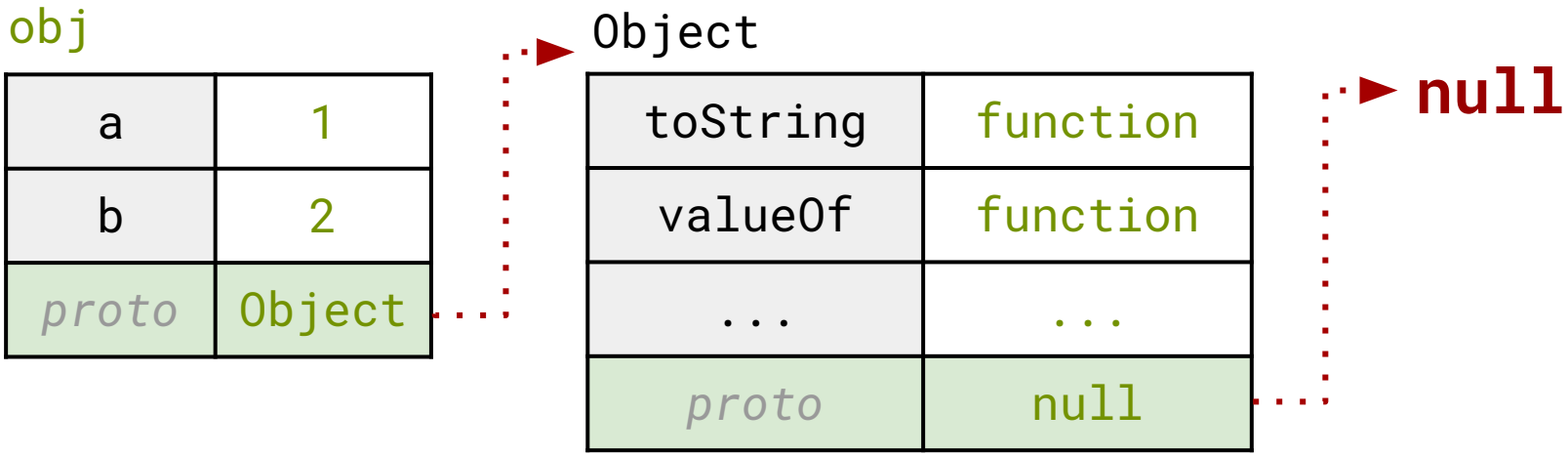
Object

```
obj.toString // [Function: toString]
```



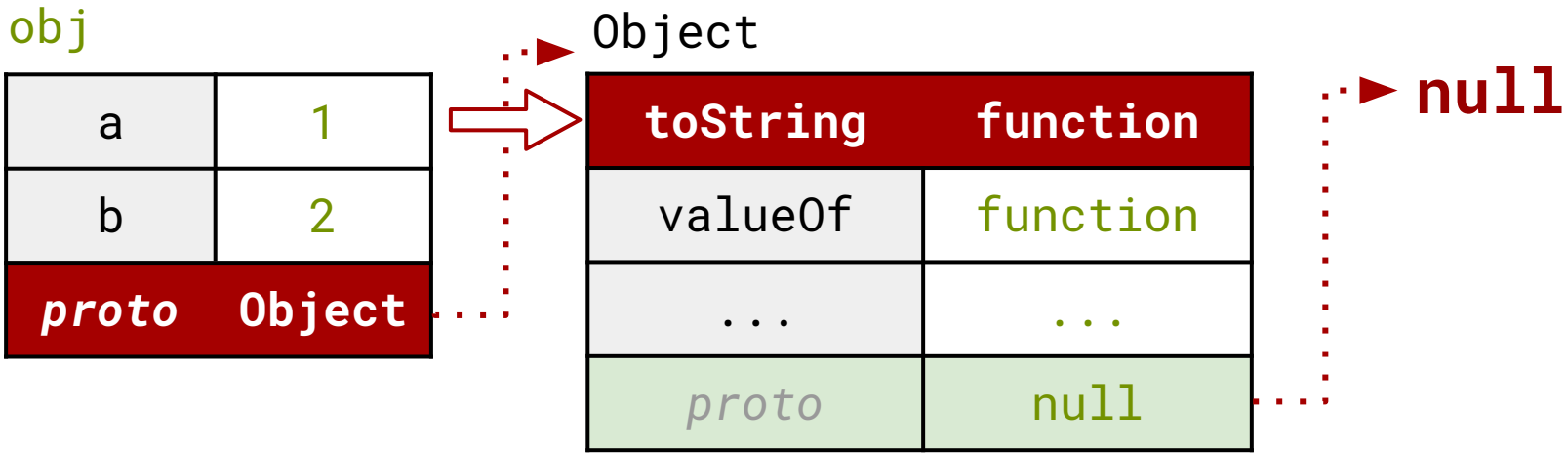
Object

```
obj.toString // [Function: toString]
```



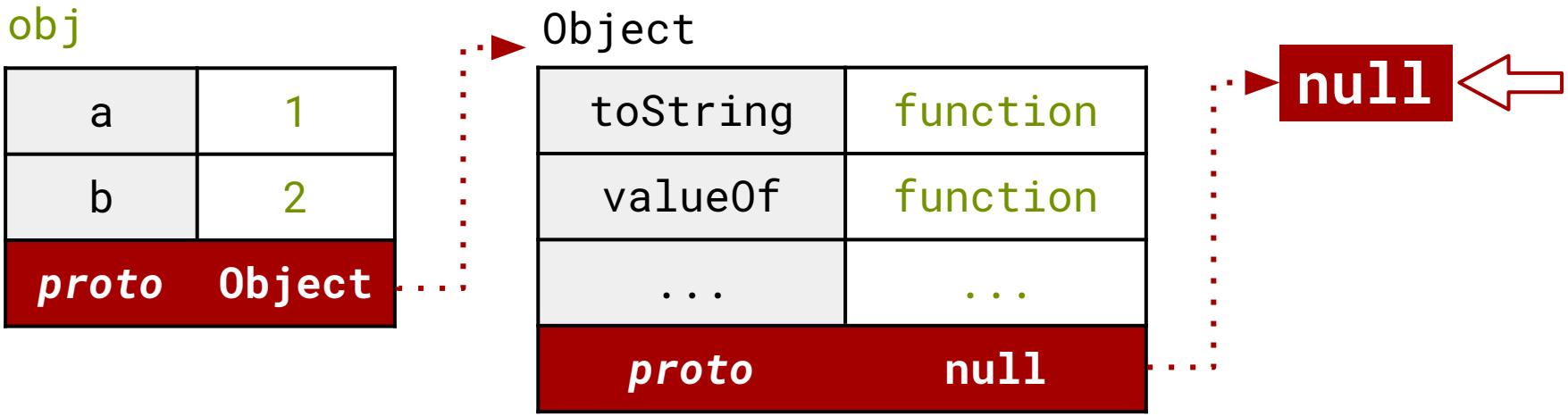
Object

```
obj.toString // [Function: toString]
```



Object

```
obj.noExiste // undefined
```

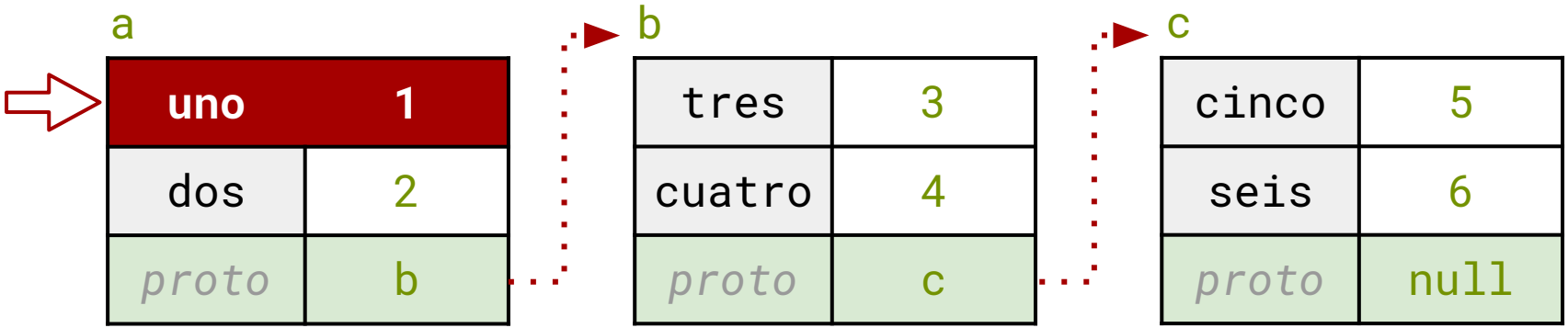


Object

- Si **A** es prototipo de **B**...
  - Todas las propiedades de **A** son visibles en **B**
  - Todas las propiedades del prototipo de **A** son visibles en **B**
  - Todas las propiedades del prototipo del prototipo de **A** son visibles en **B**
  - ....

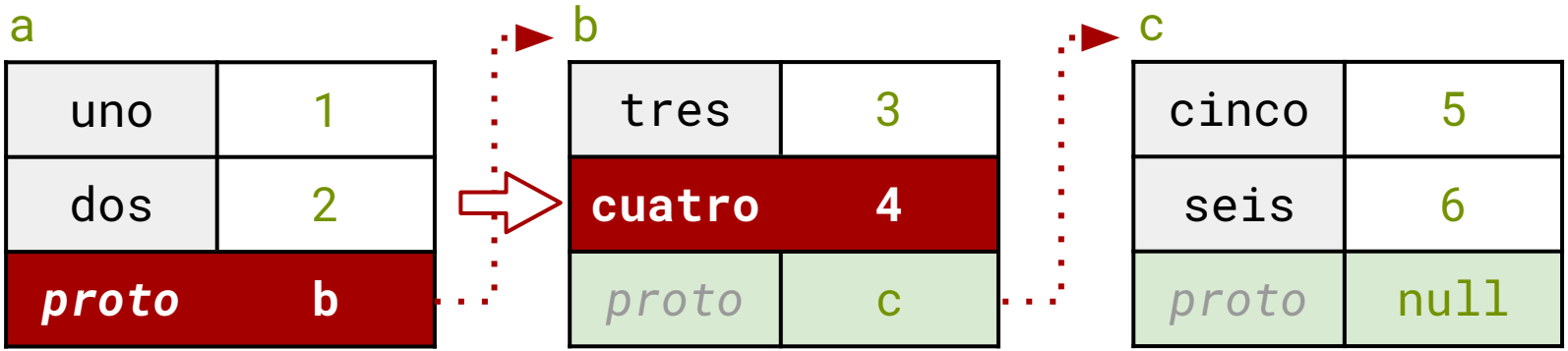
Object

```
a.uno // 1
```



Object

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



Object

```
a.cinco // 5
```

Ejercicio

- Crea un objeto **A** cuyo prototipo sea **B** cuyo prototipo sea **C** utilizando `Object.create(...)`
  - Como en el ejemplo que acabamos de ver

Ejercicio

- ¿Qué devuelve `a.toString()`?
- ¿Por qué?

Object

`obj.hasOwnProperty(prop)`

- Comprueba si la propiedad pertenece al objeto
- Útil para distinguir las propiedades heredadas

Object

```
const obj = Object.create({ a: 1 }, {
  b: { value: 2 },
  c: { value: 3, enumerable: true }
});

obj.hasOwnProperty('a'); // false
obj.hasOwnProperty('b'); // true
obj.hasOwnProperty('c'); // true
```

Object

```
const base = { common: 'uno' };

const a = Object.create(base, {
  name: { value: 'a' }
});

a.name; // 'a'
a.common; // ???
```

Object

```
base.common = 'dos';

const b = Object.create(base, {
  name: { value: 'b' }
});

b.name; // 'b'
b.common; // ???
```

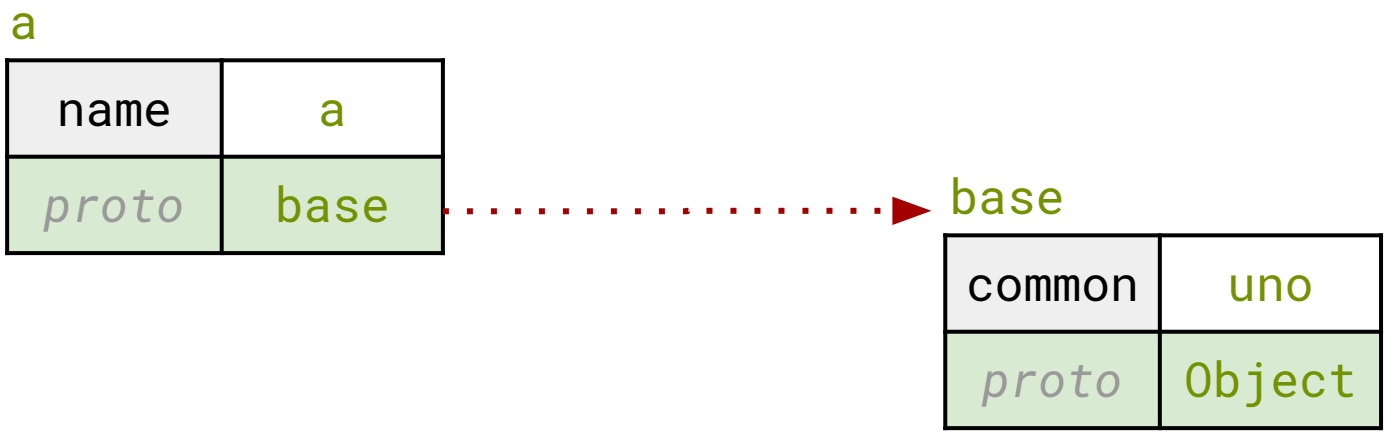
Object

```
a.common === b.common; // ???
```

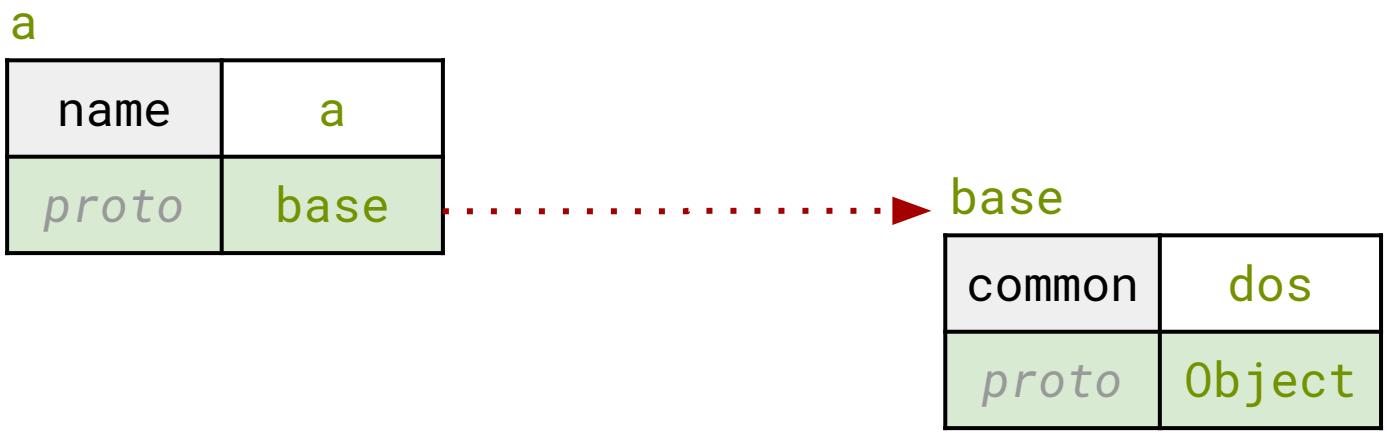
Object

```
a.common; // ???
```

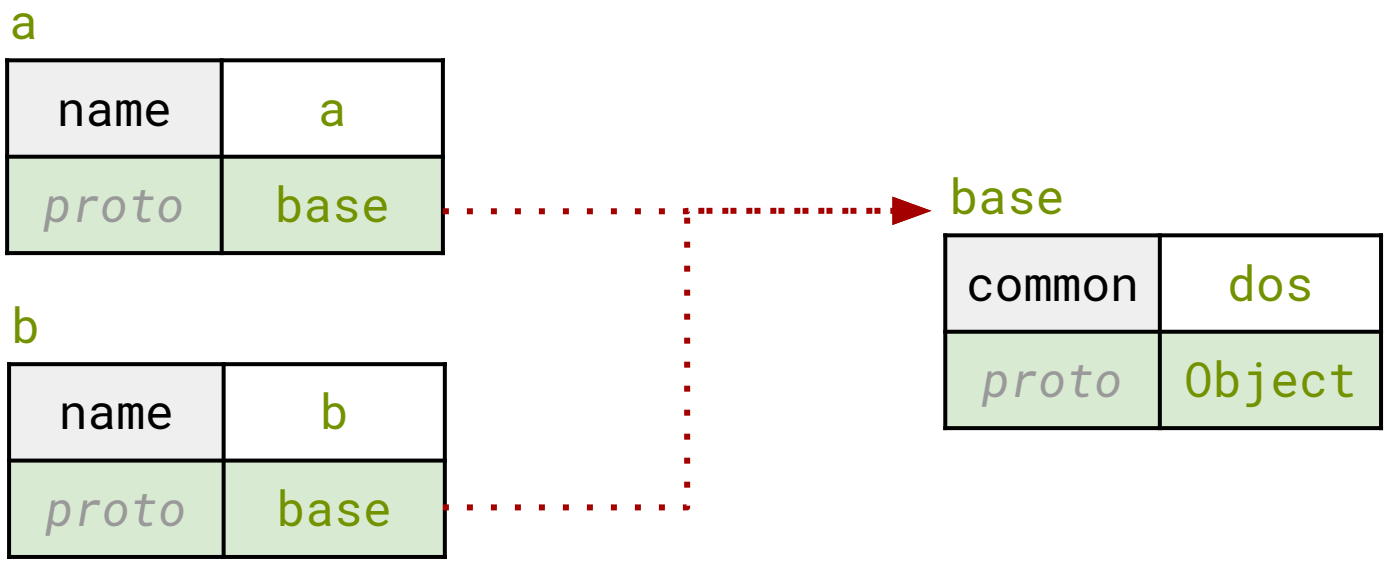
Object



Object



Object



Object

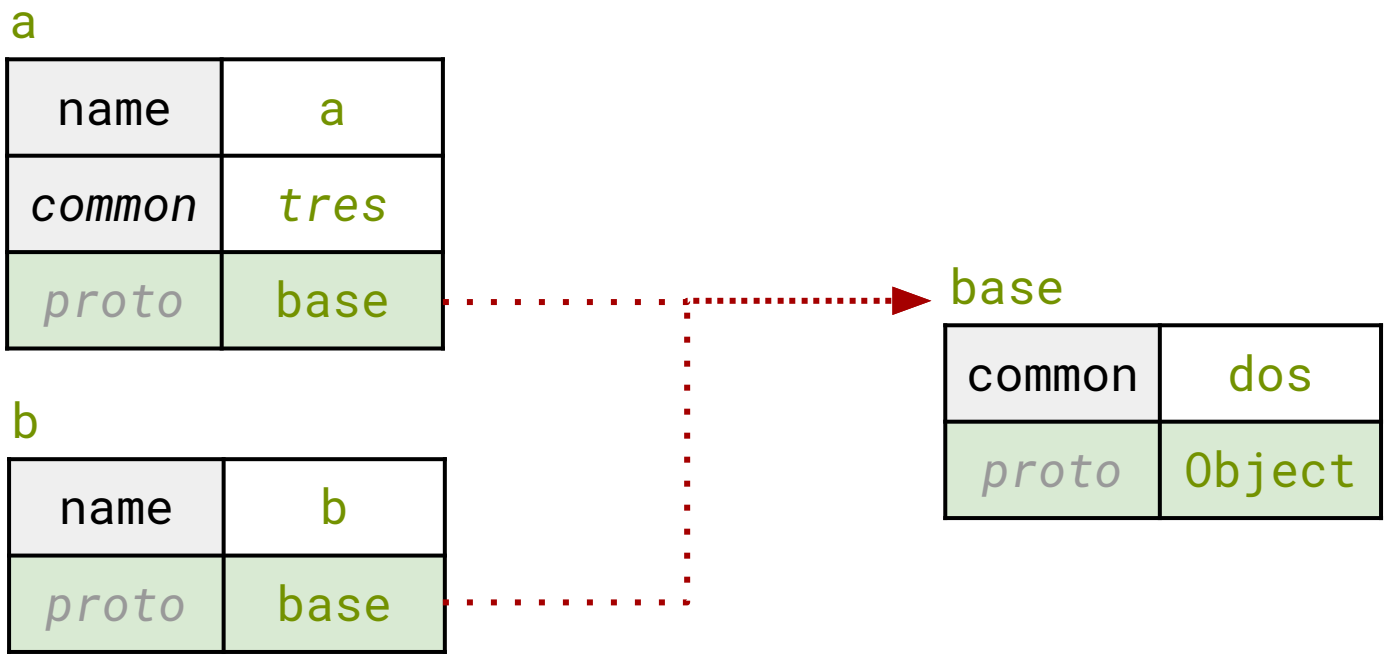
```
a.common = 'tres';
b.common; // ???
```

Object

```
a.common === b.common; // ???
```

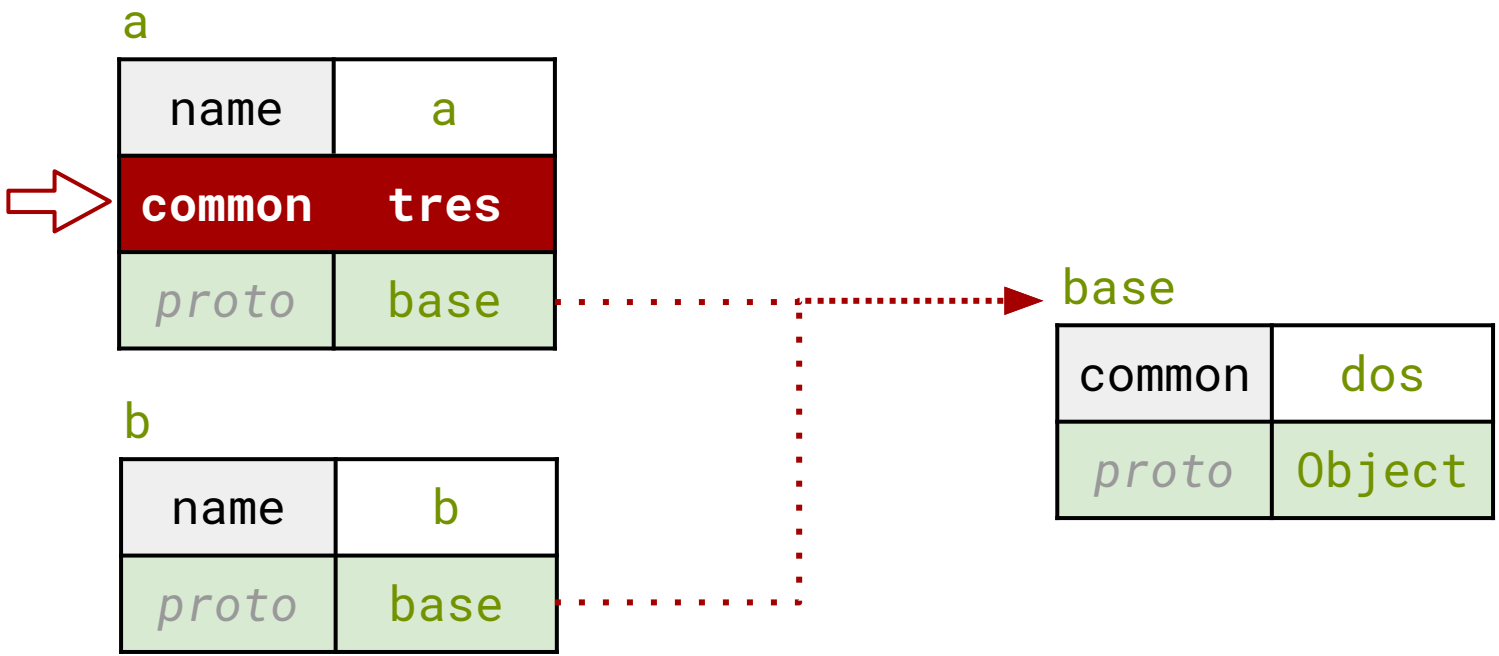
Object

```
a.common = 'tres';
```

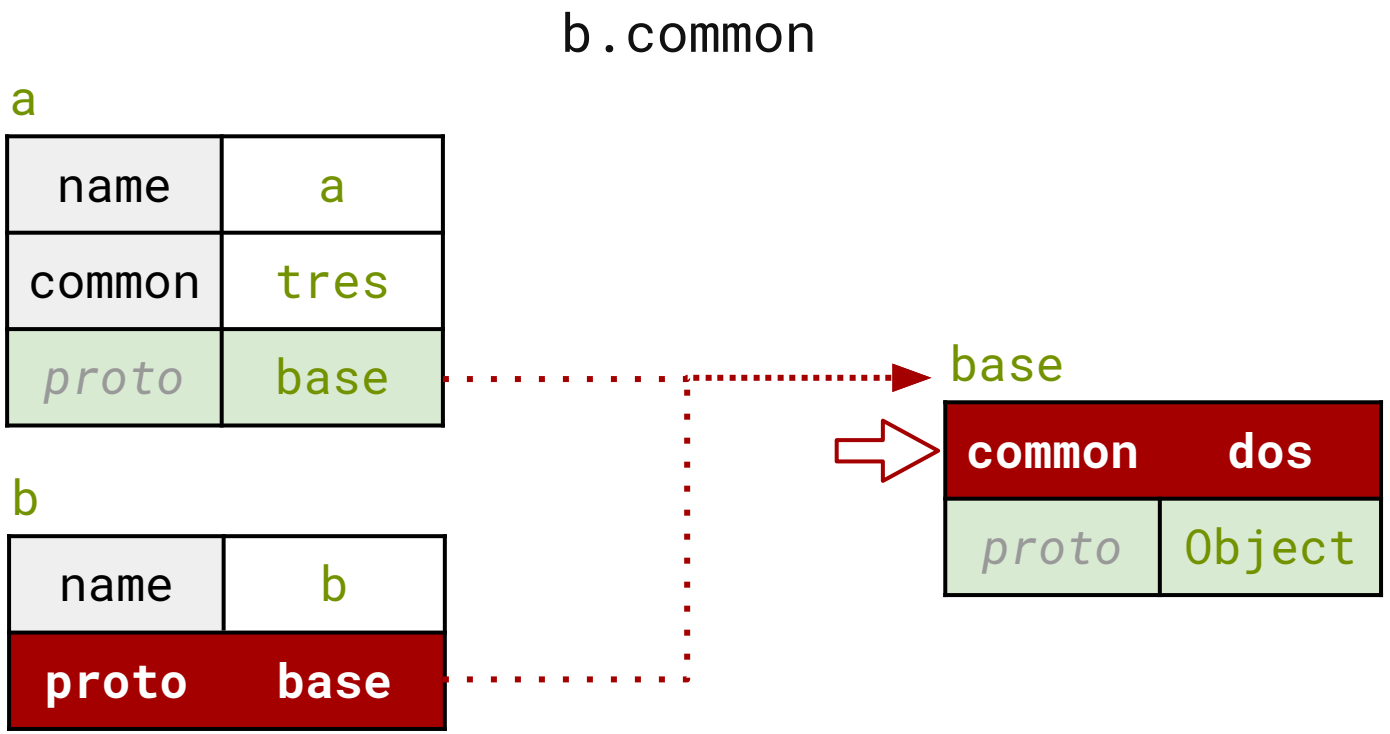


Object

a.common



Object



Object

- La cadena de prototipos es un mecanismo *asimétrico*
  - La **lectura** se propaga por la cadena
  - La **escritura** siempre es directa
- Adecuada para compartir propiedades comunes entre instancias y almacenar sólo las diferencias

Object

```
const lista = {
  items: [],
  add: function(el) { this.items.push(el); },
  getItems: function() { return this.items; }
};
```

Object

```
const todo = Object.create(lista);

todo.add('Escribir tests');
todo.add('Refactorizar el código');
todo.add('Correr los test');

todo.getItems(); // ???
```

Object

```
const compra = Object.create(lista);

compra.add('Huevos');
compra.add('Jamón');
compra.add('Leche');

compra.getItems(); // ???
```

Object

Pero... ***¿Por qué?***

Object

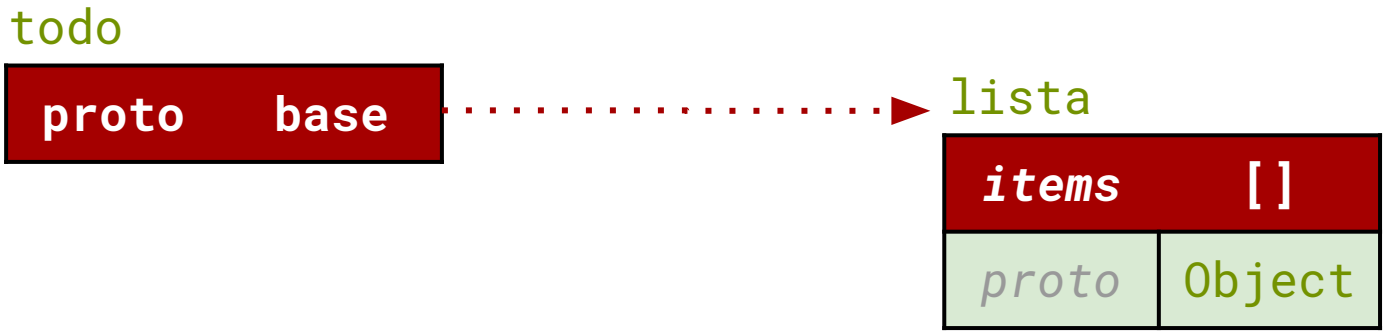
```
const todo = Object.create(lista);
```

Object

```
this.items.push(el);
```

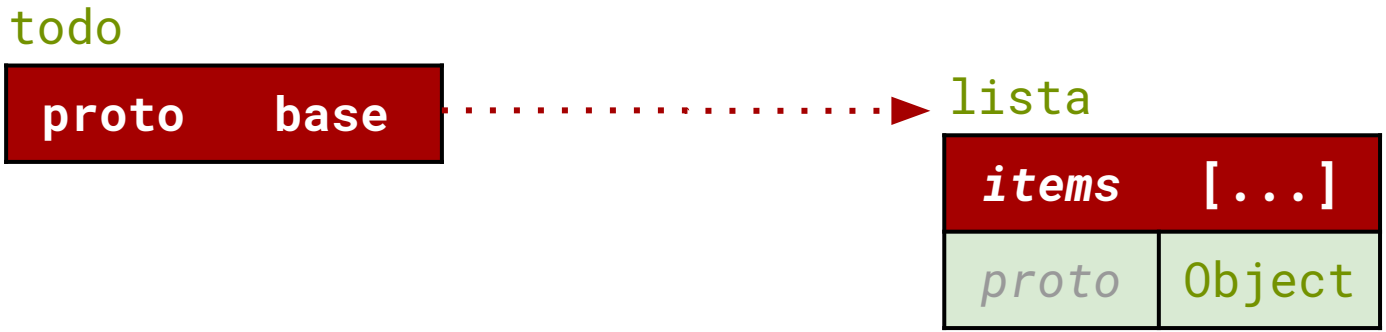
Object

```
this.items.push(el);
```



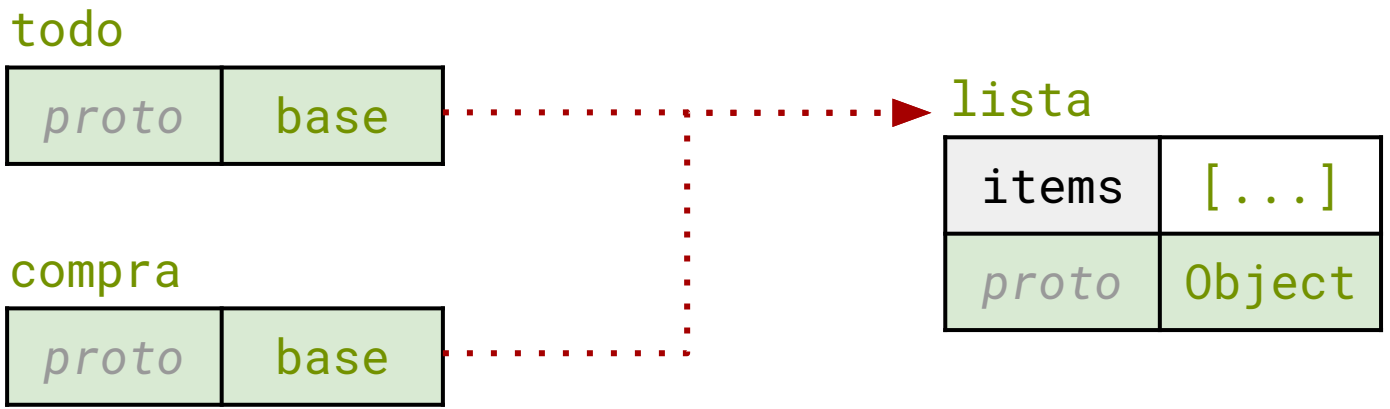
Object

```
this.items.push(el);
```



Object

```
const compra = Object.create(lista);
```



Object

```
const parent = Object.create(null, {
  x: { writable: false, value: 1 }
});

const child = Object.create(parent);

child.x = 2;

child.x; // ???
```

Object

```
const parent = Object.create({}, {
  km: { value: 0, writable: true },
  mi: {
    get: function() { return this.km / 1.60934; },
    set: function(v) { this.km = v * 1.60934; }
  }
});
```

Object

```
const child = Object.create(parent);
child.mi = 80;

child.km; // ???
parent.km; // ???
```

Funciones

Receptor

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    console.log(`Hola, ${this.nombre}`)
  }
};
```

¿Qué significa esto?

```
obj.nombre;
```

Receptor

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    console.log(`Hola, ${this.nombre}`)
  }
};
```

¿Y esto?

```
obj.saludo;
```

Receptor

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    console.log(`Hola, ${this.nombre}`)
  }
};
```

¿Y esto otro?

```
obj.saludo();
```

Receptor

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    console.log(`Hola, ${this.nombre}`)
  }
};
```

¿Es lo mismo?

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

NO

Receptor

```
obj.saludo();
```

1. ***Envía el mensaje*** “*saludo*” a obj
2. Si existe, ***obj se encarga de ejecutar*** la función adecuada
3. obj es el ***receptor***

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

1. ***Accede al valor de la propiedad*** “*saludo*” de obj
2. Supongo que es una función y ***la invoco***
3. **NO hay receptor**

Receptor

Cuatro maneras de invocar a una función:

1. **Invocación directa**

Receptor

Cuatro maneras de invocar a una función:

1. Invocación directa
2. **Enviando un mensaje a un objeto (método)**

Receptor

- El receptor del mensaje...
  - Una referencia que **no tiene binding léxico**
  - Que apunta al objeto que **recibe el mensaje**
    - *El que está a la izquierda del punto*
  - Y se **vincule** en el momento de la **invocación**

Receptor

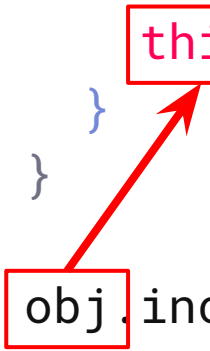
`this`

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}

obj.increment()

console.log(obj.counter)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}

obj.increment()

console.log(obj.counter)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}

obj.increment()

console.log(obj.counter)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}


obj.increment()

console.log(obj.counter)
```



Receptor

- Invocar a un método **no** es llamar a una función
  - Hay un **receptor**
    - A la izquierda del punto en **la invocación**
  - Pasos adicionales
    - **vincular `this`**
    - buscar la implementación del método

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
    console.log(`> ${this.counter}`)
  }
}

const inc = obj.increment

inc()
inc()

console.log(obj.counter)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
    console.log(`> ${this.counter}`)
  }
}

setInterval(obj.increment, 1000)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
    console.log(`> ${this.counter}`)
  }
}

const inc = obj.increment
setInterval(inc, 1000)
```

```
global.name = 'Mr. Global'

const user = {
  name: 'Ms. Property',
  greet: function() {
    console.log(`Hola, soy ${this.name}`)
  }
}

user.greet()

const greet = user.greet
greet()
```

```
const counter = {
  count: 0,
  increment: function() { this.count++; }
}

$('#button').on('click', counter.increment)
```

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    setTimeout(function() {
      console.log(`Hola, ${this.nombre}`)
    }, 100);
  }
}

obj.saludo()
```

```
function saludo() {
  console.log(`Hola, ${this.nombre}`)
}
const obj1 = {
  nombre: 'Homer'
}
const obj2 = {
  nombre: 'Fry'
}
```

Funciones

Cuatro maneras de invocar a una función:

1. Invocación directa
2. Enviando un mensaje a un objeto (método)
3. **Function.prototype**

Funciones

```
fn.call(context, arg1, arg2, …)

fn.apply(context, [arg1, arg2, …])
```

- Ejecutamos la función **fn**
- Especificando el **valor de `this` explícitamente**

```
function saludo() {
  console.log(`Hola, ${this.nombre}`)
}
const obj1 = {
  nombre: 'Homer'
}

saludo() // ???

obj1.saludo() // ???

saludo.call(obj1) // ???

setTimeout(saludo.call(obj1), 1000) // ???
```

```
const concat = [].concat

const primero = [1, 2]
const segundo = [3, 4]
const tercero = [5, 6]
```

```
[...primero, ...segundo, ...tercero]
```

```
function suma(a, b) {
  return a + b;
}

suma(1, 1) // ???

suma.call(1, 1) // ???

suma.apply([], 1, 1) // ???

suma.call(null, 1, 1) // ???

suma.apply([null, 1, 1]) // ??
```

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    console.log('Dame un segundo...')
    setTimeout(function() {
      console.log(`Hola, soy ${this.nombre}`)
    }, 1000)
  }
}

obj.saludo()
```

```
function saludo() {
  const self = this
  return function() {
    console.log(`Hola, soy ${self.nombre}`)
  }
}

const obj = { nombre: 'Homer' }

saludo(obj) // ???
saludo.call(obj) // ???
saludo.call(obj)() // ???

const fn = saludo.call(obj)
fn.call(null) // ???
fn.call({ nombre: 'Fry' }) // ??
```

¿Qué hace esta función?

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

¿Qué hace esta función?

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

const algo = misterio();

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

¿Qué hace esta función?

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

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

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

¿Qué hace esta función?

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

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

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

¿Qué hace esta función?

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

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

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

¿Qué hace esta función?

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

const obj = { nombre: 'Homer' };
const algo = misterio(obj, function() {
  return this.nombre;
});

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

¿Qué hace esta función?

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

const obj = { nombre: 'Homer' };
const algo = misterio(obj, function(saludo) {
  return `${saludo}, ${this.nombre}`;
});

algo('Hola'); // ???
```

¿Qué hace esta función?

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

const homer = { nombre: 'Homer' };
const fry = { nombre: 'Fry' };

const algo = misterio(homer, function(saludo) {
  return `${saludo}, ${this.nombre}`;
});

algo.call(fry, 'Hola'); // ???
```

¿Qué hace esta función?

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

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

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

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

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    setTimeout(bind(this, function() {
      console.log(`Hola, ${this.nombre}`)
    }), 100);
  }
}

obj.saludo()
```

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    setTimeout((function() {
      console.log(`Hola, ${this.nombre}`)
    }).bind(this), 100);
  }
}

obj.saludo()
```

arrow functions

Arrow functions

- Sintaxis alternativa para definir funciones anónimas
  - Más corta
  - Más conveniente
  - Más segura

Arrow functions

```
(arg1, arg2, …) => { statement; statement; return …; }
```

Arrow functions

```
const sum = (a, b) => {
  const result = a + b;
  return result;
};
```

Arrow functions

```
const sum = (a, b) => {
  const result = a + b;
  return result;
};
```

Arrow functions

```
const sum = (a, b) => {
  const result = a + b;
  return result;
};
```

Arrow functions

```
const sum = (a, b) => {
  const result = a + b;
  return result;
};
```

Arrow functions

```
(arg1, arg2, …) => { statement; statement; return …; }

(arg1, arg2, …) => expression;
```

Arrow functions

```
const sum = (a, b) => a + b;
```

Arrow functions

```
const sum = (a, b) => { return a + b; };
```

Arrow functions

```
const sum = (a, b) => ({ result: a + b });
```

Arrow functions

```
(arg1, arg2, …) => { statement; statement; return …; }

(arg1, arg2, …) => expression;

arg => expression;
```

Arrow functions

```
const random = n => Math.floor(Math.random() * n);
```

```
const obj = {
  nombre: 'Homer',
  saludo: () => console.log(`Hola, soy ${this.nombre}`)
}

obj.saludo() // ???
```

```
const obj = {
  nombre: 'Homer',
  generarSaludo: function(saludo) {
    return () => {
      console.log(`${saludo}, soy ${this.nombre}`)
    }
  }
}

const sp = obj.generarSaludo('Hola')
sp() // ???
```

```
function doble(a) {
  return a * 2
}

function suma(a, b) {
  return a + b
}

function masOMenos(a, b) {
  if (Math.random() < .5) {
    return a - b
  } else {
    return a + b
  }
}
```

```
const saludo = () => {
  console.log(`Hola, soy ${this.nombre}`)
}

const obj = { obj: 'Homer' }

const binded = saludo.bind(obj)

binded() // ???
```

```
const generator = {
  name: 'User Generator',
  createUser: function(name) {
    return { name, greet: () => console.log(`Hola, soy ${this.name}`) }
  }
}

const homer = generator.createUser('Homer')
```

Clausuras

```
let a = 1

function what() {
  return a
}
```

redradix

```
function what2() {
  {
    let a = 1
  }
  return a
}
```

```
function what3() {
  let a = 1
  return function() {
    return a
  };
}

let thing1 = what3()
```

```
function what3() {
  let a = 1
  return function() {
    return a
  };
}

let thing1 = what3()
console.log(thing1())
```

Clausuras

```
function counter() {
  return () => {
    let i = 0;
    return i++;
  };
}
```

Clausuras

```
const c1 = counter();
```

Clausuras

```
const c1 = counter();
console.log(c1());
```

Clausuras

```
const c1 = counter();
console.log(c1()); // 0
```

Clausuras

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1());
```

Clausuras

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 0
console.log(c1()); // 0
```

Clausuras

```
const c1 = counter();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

Clausuras

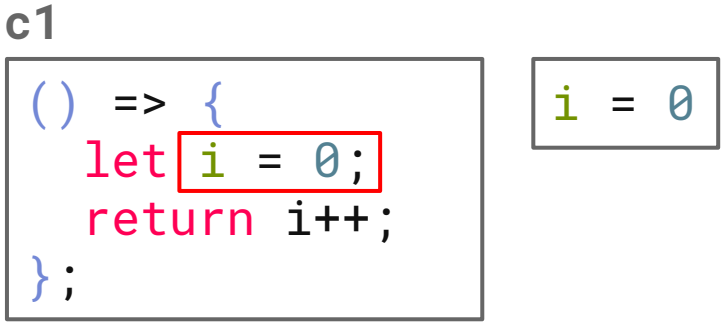
```
const c1 = counter();
c1();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

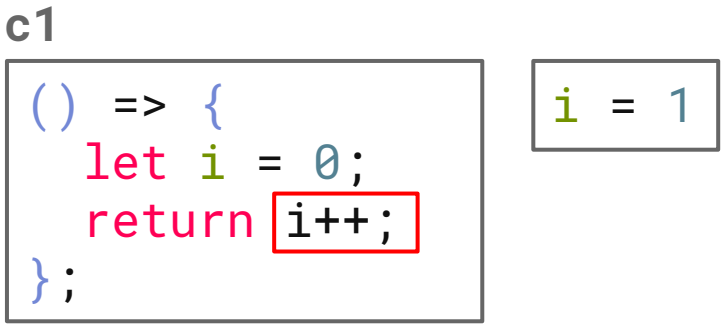
Clausuras

```
const c1 = counter();
c1();
```



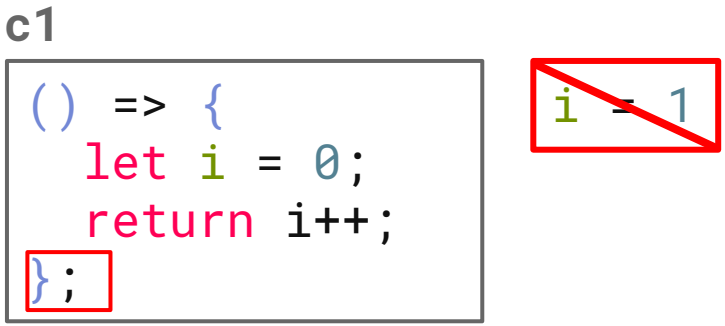
Clausuras

```
const c1 = counter();
c1(); // 0
```



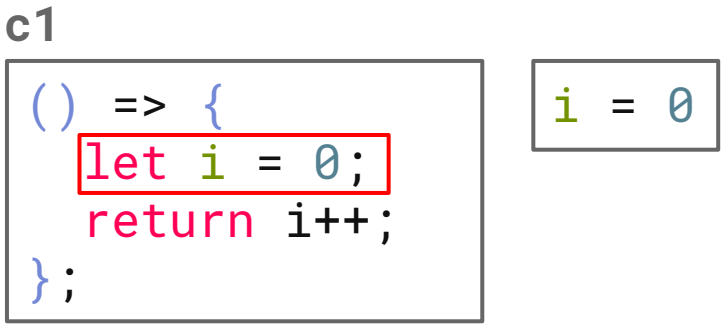
Clausuras

```
const c1 = counter();
c1(); // 0
```



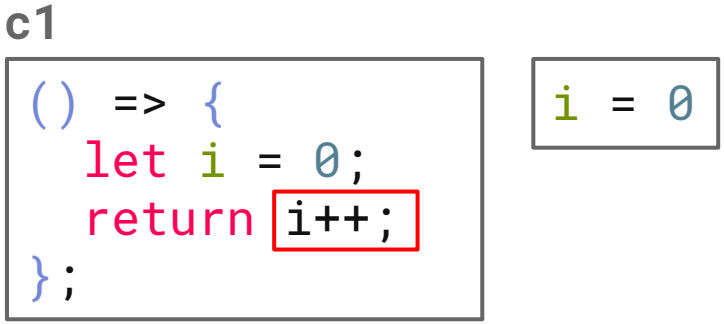
Clausuras

```
const c1 = counter();
c1(); // 0
c1();
```



Clausuras

```
const c1 = counter();
c1(); // 0
c1(); // 0
```



Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Clausuras

```
function counter() {
  let i = 0;
   return () => i++;
}
```

Clausuras

```
function counter() {
  let i = 0;
  return () => {
    i++;
    return i;
  };
}
```

Clausuras

```
const c1 = counter();
console.log(c1());
```

Clausuras

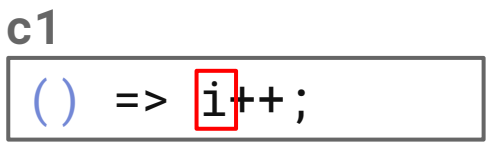
```
const c1 = counter();
console.log(c1()); // 0
console.log(c1());
```

Clausuras

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 1
console.log(c1()); // 2
```

Clausuras

```
const c1 = counter();
```

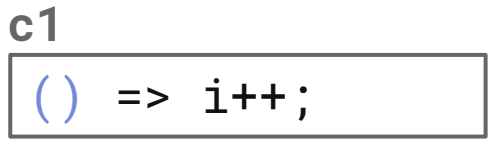


Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Clausuras

```
const c1 = counter();
let i = 10;
c1();
```



Clausuras

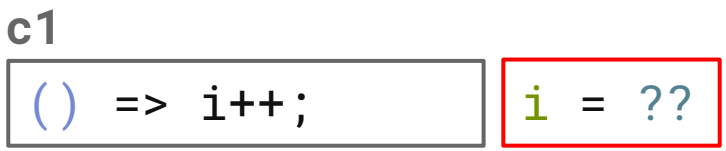
```
const c1 = counter();
let i = 10;
c1();              // ???
console.log(i); // ???
```

c1

```
() => i++;
```

Clausuras

```
const c1 = counter();
let i = 10;
c1();              // 0
console.log(i); // 10
```

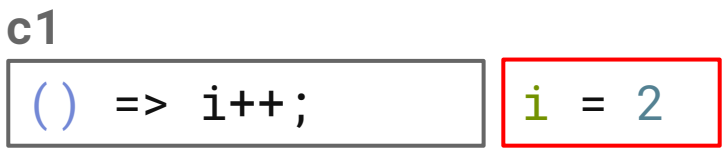


Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

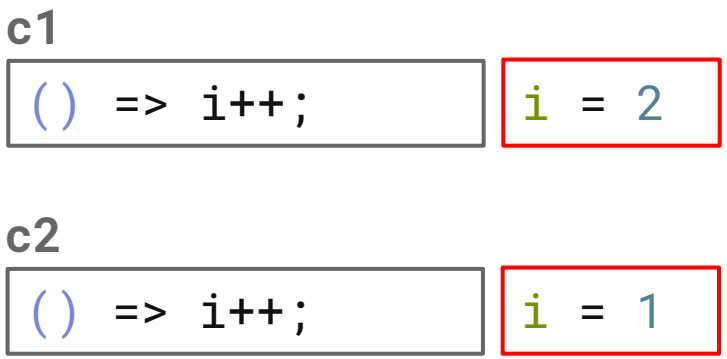
Clausuras

```
const c1 = counter();
c1(); // 0
c1(); // 1
```



Clausuras

```
const c1 = counter();
c1(); // 0
c1(); // 1

const c2 = counter();
c2(); // ???
```



redradix

Clausuras

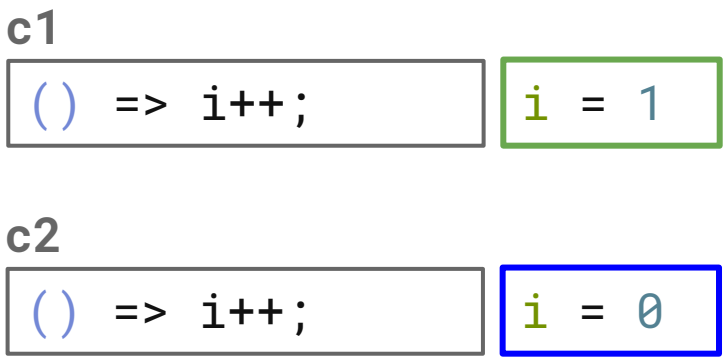
- Las variables en javascript tienen *alcance indefinido*
  - Persisten **durante todo el tiempo que haga falta**
  - Solo se destruyen cuando **es imposible acceder a ellas**
- Una variable libre mantiene viva la variable a la que hacía referencia en el contexto en el que fue definida
- Este fenómeno se denomina *clausura*

Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Clausuras

```
const c1 = counter();
c1(); // 0
c1(); // 1

const c2 = counter();
c2(); // 0
```



redradix

Constructores

Constructores

Cuatro maneras de invocar a una función:

1. Invocación directa
2. Enviando un mensaje a un objeto (método)
3. Function.prototype
4. **new**

Constructores

- Una función se ejecuta como constructor cuando la llamada está precedida por **new**
- Antes de ejecutar un constructor suceden **tres cosas**:

Constructores

1. Se crea **un nuevo objeto** vacío
2. Se le asigna como **prototipo** el **valor de la propiedad `prototype`** del constructor
3. **`this`** dentro del constructor se vincula a este nuevo objeto

Constructores

- Por último, se ejecuta el código del constructor
- El valor de la expresión **`new Constructor()`** será:
  - El nuevo objeto...
  - ...a no ser que el constructor devuelva otro valor con **`return`**

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

Constructores

¿Verdadero o falso?

```
toby.hasOwnProperty("name")
```

Constructores

¿Verdadero o falso?

```
toby.hasOwnProperty("sit")
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

Constructores

- Cada instancia guarda su propio estado
- Pero comparten la implementación de los métodos a través de su prototipo

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");

Dog.prototype.sit = function() {
  console.log(`* ${this.name} does not understand.`);
}

const spot = new Dog("Spot");
spot.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");

Dog.prototype.sit = function() {
  console.log(`* ${this.name} does not understand.`);
}

const spot = new Dog("Spot");
spot.sit();
toby.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.sit = () => {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

Ejercicio

- Escribe un constructor **User** que reciba un nombre como parámetro, lo guarde en una propiedad y tenga un método **greet** que muestre un saludo con su nombre

Ejercicio

- Escribe un constructor **Root** de tal manera que solo **se pueda instanciar una vez**

```
function User(name) {
  this.name = name
  this.usersCreated++
}

User.prototype = {
  greet: function() {
    console.log(`Hola, soy ${this.name}`)
  },
  getTotalUsers: function() {
    return this.usersCreated
  },
  usersCreated: 0
}
```

```
function User(name) {
  this.name = name
}

User.prototype = {
  greet: function() {
    console.log(`Hola, soy ${this.name}`)
  }
}

const homer = new User('Homer')
const fry = new User('Fry')
```

```
const homer2 = new User('Homer')
console.log(homer === homer2) // ???

console.log(homer.greet === homer2.greet) // ???

homer2.greet = () => console.log('Buen dia')

homer.greet() // ??? (why??)

User.prototype.greet = () => console.log('Hola!')
fry.greet() // ???
homer2.greet() // ???
```

Ejercicio

- Escribe la función **myNew** que replique el comportamiento de **new** utilizando **Object.create**

Ejercicio

- Escribe la función **withCount**
  - *(siguiente diapositiva)*

```
function User(name) {
  this.name = name
}

User.prototype = {
  greet: function() {
    console.log(`Hola, soy ${this.name}`)
  }
}

const CountedUser = withCount(User);
const u1 = new CountedUser('Homer')
const u2 = new CountedUser('Fry')

u1.greet() // 'Hola, soy Homer'

CountedUser.getInstanceCount() // 2
```

```
function Animal(species, color) {
  this.species = species
  this.color = color
}

Animal.prototype = {
  toString: function() {
    return `Un ${this.species} de color ${this.color}`
  },
  getSpecies() {
    return this.species
  }
}
```

```
function Dog(color, name) {
  this.name = name
  // ???
}

Dog.prototype = {
  toString: function() {
    // ???
  }
}

var toby = new Dog('moteado', 'Toby');
toby.getSpecies() // 'perro'
toby.toString() // 'Un perro de color moteado que se llama Toby'
```

```
console.log(toby instanceof Perro) // ???
console.log(toby instanceof Animal) // ???
console.log(toby instanceof Object) // ???

console.log(Perro instanceof Animal) // ???
console.log(Perro instanceof Function) // ???
```

Ejercicio

- Partiendo de la clase **`Container`**...
  - Escribe dos constructores derivados:
    - **`ItemContainer`**
    - **`NestedContainer`**

```
function Container(name) {
  this.name = name
}

Container.prototype = {
  canFit: function(item) {
    throw new Error('Abstract method')
  },
  store: function(item) {
    throw new Error('Abstract method')
  },
  retrieve: function(index) {
    throw new Error('Abstract method')
  }
}
```

Ejercicio

- **`ItemContainer(`*`name`*`)`**
  - Hereda de **`Container`**
  - Contenedor de **`Items`**
  - Implementa los métodos abstractos de **`Container`**
  - Puede contener infinitos **items**

```
function Item(name, size, category, createdAt) {
  Object.assign(this, { name, size, category, createdAt })
}

Item.prototype.getSize = function() { return this.size }
```

```
const itemContainer = new ItemContainer('Test Container')

const item1 = new Item('Item1', 10, 'test', new Date())

itemContainer.canFit(item1) // true
const index = itemContainer.store(item1)
console.log(index) // [0]

const retrieved = itemContainer.retrieve(index)
console.log(retrieved.name) // Item1
```

Ejercicio

- **`ItemBox(`*`capacity`*`)`**
  - Hereda de **`ItemContainer`**
  - Tiene un tamaño limitado
    - Parámetro del constructor
  - Cada **`item`** que se guarda ocupa espacio
    - Propiedad **.size**
  - La suma de los tamaños de los **`items`** que aloja no puede exceder su capacidad

```
const box = new ItemBox(10)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())

box.store(item1)
box.store(item2)

box.canFit(item3) // false

console.log(box.retrieve([1]).name) // Item 2
```

Ejercicio

- **`NestedContainer(`*`name, subcontainers`*`)`**
  - Hereda de **`Container`**
  - Contenedor de **`Containers`**
  - Implementa los métodos abstractos de **`Container`**
  - Recibe los sub-contenedores en el constructor

Ejercicio

- `NestedContainer(`*`name`*`, `*`subcontainers`*`)`
  - `store(item)`
    - Delega en el primer sub-container en el que quepa **item**
  - `canFit(item)`
    - ¿Cabe **item** en algún sub-container?
  - `retrieve(index)`
    - **index** es un array de múltiples elementos
    - El primer elemento es el índice del sub-container

```
const boxes = [new ItemBox(10), new ItemBox(10)]
const nestedContainer = new NestedContainer('NestedContainer', boxes)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
```

```
nestedContainer.store(item1)
const i1 = nestedContainer.store(item2)
console.log(i1) // [0, 1]

nestedContainer.canFit(item3) // true
const i2 = nestedContainer.store(item3)

console.log(i2) // [1, 0]

nestedContainer.canFit(item4) // false

console.log(nestedContainer.retrieve([0, 1]).name) // Item 2
```

Ejercicio

- Partiendo de **`NestedContainer`**…
  - **`Shelf`**
    - Conjunto de **`ItemBoxes`**
  - **`Rack`**
    - Conjunto the **`Shelf`**
  - **`Warehouse`**
    - Conjunto de **`Rack`**

Ejercicio

- `Shelf(`*`maxBoxes, boxCapacity`*`)`
  - Empieza *vacía* (cero cajas)
  - Las cajas se van creando cuando sea necesario

```
const shelf = new Shelf(2, 10)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

// shelf starts with 0 boxes...
console.log(shelf.subcontainers.length) // 0

// ...but has to create a new box to hold item1
shelf.canFit(item1) // true
shelf.store(item1)
console.log(shelf.subcontainers.length) // 1
```

```
shelf.canFit(item2) // true
shelf.store(item2)
console.log(shelf.subcontainers.length) // 1

shelf.canFit(item3) // true
shelf.store(item3)
console.log(shelf.subcontainers.length) // 2

shelf.canFit(item4) // false

shelf.canFit(item5) // true
console.log(shelf.store(item5)) // [0, 2]
```

Ejercicio

- `Rack(numShelves, boxesPerShelf, boxCapacity)`
  - Empieza con **numShelves** instancias of **Shelf** vacias
    - las genera en el constructor

```
const rack = new Rack(2, 2, 5)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

console.log(rack.subcontainers.length) // 2

rack.store(item1)
rack.store(item2)
console.log(rack.store(item3)) // [1, 0, 0]

rack.canFit(item4) // false
rack.canFit(item5) // true

console.log(rack.retrieve([0, 1, 0]).name) // Item 2
```

Ejercicio

- `Warehouse(racks)`
  - Recibe una *configuración de Racks*
  - Su peculiaridad:
    - Comprueba que un elemento quepa antes de insertarlo
    - en el método **.store(...)**
    - Levanta excepción si no cabe

```
const warehouse = new Warehouse([
  new Rack(2, 2, 5),
  new Rack(2, 1, 10)
])

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

console.log(warehouse.store(item1)) // [0, 0, 0, 0]
warehouse.store(item2)
warehouse.store(item3)

warehouse.canFit(item4) // true
console.log(warehouse.store(item4)) // ????

console.log(warehouse.retrieve([0, 0, 1, 0]).name) // Item 2
```

```
const warehouse = new Warehouse([
  new Rack(2, 2, 5),
  new Rack(2, 1, 10)
])

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

console.log(warehouse.store(item1)) // [0, 0, 0, 0]
warehouse.store(item2)
warehouse.store(item3)

warehouse.canFit(item4) // true
console.log(warehouse.store(item4)) // ????

console.log(warehouse.retrieve([0, 0, 1, 0]).name) // Item 2
```

Clases

```
class User {
  constructor(name) {
    this.name = name
  }

  greet() {
    console.log(`Hola, soy ${this.name}`)
  }
}
```

```
class Root extends User {
  constructor() {
    // OBLIGATORIO llamar a super desde el constructor
    super('ROOT')
  }
  greet() {
    super.greet()
  }
}
```

```
class Root extends User {
  constructor() {
    // OBLIGATORIO llamar a super desde el constructor
    super('ROOT')
  }
  greet() {
    super.greet()
  }
}
```

Ejercicio

- Reescribe el ejercicio anterior con **Animal** y **Dog** utilizando **class** y **extend**

Ejercicio

- Traduce los constructores de la simulación del almacén a clases
  - **Warehouse**, **Rack**, **Shelf**, **ItemBox**, **ItemContainer**, **NestedContainer**, **Item** y **Container**

```
class User {
  constructor(name) {
    this.name = name
  }
  greet() {
    console.log(`Hola, soy ${this.name}`)
  }
}

const u1 = new User('Homer')
const u2 = new User('Fry')
```

```
u1.greet.call(u2) // ???

u2.greet = u1.greet

u2.greet() // ???

User.prototype.greet = () => console.log('How do you do?')

u1.greet() // ???

u2.greet() // ???
```

Clases Anónimas

- Se puede utilizar **class** como expresión
- Permite crear clases dinámicas y/o anónimas

Clases Anónimas

```
const Mammal = class {
  constructor(name) {
    this.name = name;
  }
}

const buddy = new Mammal('Buddy');
console.log(buddy.name)
```

Ejercicio

- Reescribe **withCount** para clases

Iterators

Iterators

- Interfaz (`iterable protocol`)
- Cómo recorrer los elementos de una colección
  - cualquier colección, no sólo `Array`
- Integración con el lenguaje
  - `for...of`
  - `Array.from(...)`

Iterators

- Cualquier objeto
- Con un método `.next()`
- Que devuelve *un objeto* con dos propiedades:
  - `value`
  - done

Iterators

```
let i = 0;

const iterator = {
  next: () => {
    return { done: false, value: i++ };
  }
}
```

Iterators

```
function makeIterator(array) {
  let i = 0;
  return {
    next: () => {
      const done = i === array.length;
      return { done, value: (done || array[i++]) };
    }
  };
}
```

Iterators

```
function makeIterator(array) {
  let i = 0;
  return {
    next: () => {
      const done = i === array.length;
      return { done, value: (done || array[i++]) };
    }
  };
}
```

Iterators

```
function makeIterator(array) {
  let i = 0;
  return {
    next: () => {
      const done = i === array.length;
      return { done, value: (done || array[i++]) };
    }
  };
}
```

Iterators

```
let i = makeIterator([1, 2, 3, 4]);

console.log(i.next()); // { value: 1, done: false }
console.log(i.next()); // { value: 2, done: false }
console.log(i.next()); // { value: 3, done: false }
console.log(i.next()); // { value: 4, done: false }
console.log(i.next()); // { value: true, done: true }
```

Iterators

```
let i = makeIterator([1, 2, 3, 4]);

let next = i.next();
while (!next.done) {
  console.log(next.value);
  next = i.next();
}
```

Iterators

```
let i = makeIterator([1, 2, 3, 4]);

for (let n = i.next(); !n.done; n = i.next()) {
  console.log(n.value);
}
```

Iterators: Ejercicio

- Implementa un iterador que...
  - ...devuelva los elementos de un array en orden aleatorio
  - ...devuelva los números de un rango especificado
  - ...devuelva la serie de fibonacci (infinita!)

Iterators

- Un **iterable** es una estructura de datos que...
  - Su propiedad `[Symbol.iterator]` es una función
  - Devuelve un iterador
  - Que recorre los datos de la estructura

Iterators

- Javascript sabe cómo recorrer **iterables**
  - `for … of`
  - `Array.from(…)`
- Muchos objetos nativos son iterables
  - `Array`
  - `Map`
  - `...`

Iterators

```
const list = [1, 2, 3, 4];

for (const item of list) {
  console.log(item);
}
```

Iterators

- Para crear nuestros propios **iterables:**
  - Metemos en `[Symbol.iterator]`...
  - una función
  - que devuelva un **iterador** que recorra la colección

Ejercicio

- Haz que **ItemContainer** sea **iterable**
  - Devuelve cada uno de los **items** contenidos

```
const box = new ItemBox(10)

box.store(new Item('Item 1', 3, 'test', new Date()))
box.store(new Item('Item 2', 3, 'test', new Date()))
box.store(new Item('Item 3', 1, 'test', new Date()))

for (const item of box)
  console.log(item.name) // logs every item name
```

Ejercicio

- Haz que **NestedContainer** sea **iterable**
  - Devuelve cada uno de los **items** contenidos...
  - ... en **cada uno de sus contendores**!

```
const warehouse = new Warehouse([
  new Rack(2, 2, 5),
  new Rack(2, 1, 10)
])

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

warehouse.store(item1)
warehouse.store(item2)
warehouse.store(item3)
warehouse.store(item4)
warehouse.store(item5)

for (const item of warehouse)
  console.log(item.name) // logs every item name
```

Iterators

```
let i = {
  [Symbol.iterator]: () => makeIterator([1, 2, 3, 4])
};
```

Iterators

```
for (const v of i) {
  console.log(v);
}
```

Generators

Generators

- Una *función especial*
- Que se comporta como una factoría de iteradores
  - Al ejecutarse devuelve un iterador
  - Simplifica la escritura de iteradores
  - Por cómo gestiona el estado de la iteración

Generators

- Sintaxis dedicada:
  - `function*`
  - `yield`

redradix

Generators

```
function* generator() {
  yield 1
  yield 2
}
```

Generators

```
const i = generator()
```

Generators

```
const i = generator()

let n = i.next()
console.log(n.value) // 1
```

redradix

Generators

```
const i = generator()

let n = i.next()
console.log(n.value) // 1

n = i.next()
console.log(n.value) // 2
```

Generators

```
const i = generator()

let n = i.next()
console.log(n.value) // 1

n = i.next()
console.log(n.value) // 2

n = i.next()
console.log(n.value) // undefined
console.log(n.done) // true
```

redradix

Generators

```
function* generator() {
  yield 1
  return 2
}
```

Generators

```
function* range(from, to) {
  for (let i = from; i < to; i++) {
    yield i
  }
}
```

Generators

```
function* range(from, to) {
  for (let i = from; i < to; i++) {
    yield i
  }
}
```

Generators

```
function* range(from, to) {
  for (let i = from; i < to; i++) {
    yield i
  }
}
```

Generators

```
for (let n of range(10, 20))
  console.log(n);
```

redradix

Generators

```
function* peculiar() {
  console.log('Give me a 1!');
  yield 1;
  console.log('Give me a 2!');
  yield 2;
  console.log('Give me a 3!');
  yield 3;
}
```

Generators

```
const i = peculiar();
```

Generators

```
const i = peculiar();

let n = i.next(); // Va un uno!
console.log(n.value); // 1
```

Generators

```
const i = peculiar();

let n = i.next(); // Va un uno!
console.log(n.value); // 1

n = i.next(); // Va un dos!
console.log(n.value); // 2
```

Ejercicio

- Haz que **NestedContainer** sea **iterable**
  - Utilizando un **generador**
  - Una pista:
    - la ejecución de un *generador* devuelve un ***iterador***

Sets

Sets

`new Set(`*`iterable`*`)`

- Almacena valores únicos
  - Primitivos
  - Por referencia (`object`)

Sets

```
const s = new Set();
s.add('A');
console.log(s.has('A')); // true
console.log(s.has('B')); // false
```

Sets

- `add(value)`
- `delete(value)`
- `clear()`
- `has(value)`

Sets

```
const s2 = new Set(['A', 'B']);
console.log(Array.from(s2));
```

Sets

```
const s2 = new Set(['A', 'B']);

for (let value of s2) {
  console.log(value);
}
```

Sets

```
const s2 = new Set(['A', 'B']);

for (let value of s2) {
  console.log(value);
}
```

Ejercicio

- Implementa las tres operaciones fundamentales
  - `union(A, B)`
  - `intersection(A, B)`
  - `difference(A, B)`

Ejercicio

```
> const t1 = new Set(['A', 'B'])
> const t2 = new Set(['C', 'B'])
> union(t1, t2) // Set { 'A', 'B', 'C' }
> intersection(t1, t2) // Set { 'B' }
> difference(t1, t2) // Set { 'A' }
> difference(t2, t1) // Set { 'C' }
```

Maps

Maps

`new Map(`*`iterable`*`)`

- Almacenar pares clave-valor
- Diccionarios
- No son un tipo nativo
  - **`typeof`** nos dice que son `'object'`

Maps

```
const m = new Map();
m.set('clave', 'valor');
console.log(m.get('clave'));
```

Maps

```
const m = new Map();
m.set('clave', 'valor');
console.log(m.get('clave'));
```

Maps

```
const m = new Map([['a', 1], ['b', 2]]);
```

Maps

```
const m = new Map([['a', 1], ['b', 2]]);
console.log(Array.from(m));
```

Maps

- `.set(key, value)`
- `.get(key)`
- `.has(key)`
- `.delete(key)`
- `.clear()`
- `.size`

Maps

```
const m = new Map([['a', 1], ['b', 2]]);

console.log(m.has('a')); // true
console.log(m.has('c')); // false

m.delete('b'); // true
m.delete('c'); // false

console.log(m.get('b')); // undefined
console.log(m.size); // 1
```

Maps

Para recorrer un mapa...

- `.keys()`
- `.values()`
- `.forEach(`*`fn`*`)`
- `.entries()`
- La instancia es iterable

Maps

```
const m = new Map([['a', 1], ['b', 2]]);

console.log(Array.from(m.keys())); // [ 'a', 'b' ]
console.log(Array.from(m.values())); // [ 1, 2 ]
```

Maps

```
const m = new Map([['a', 1], ['b', 2]]);

m.forEach(function(valor, clave) {
  console.log(clave + ' -> ' + valor);
});

// a -> 1
// b -> 2
```

Maps

`Map` > `Object`

- Mejor semántica
  - API más limpia
  - Intención del autor más clara

Maps

`Map` > `Object`

- No afecta la herencia de prototipos
  - Los pares de un mapa siempre son de ese mapa

Maps

`Map` > `Object`

- Conservan el orden de inserción de los pares
  - Los objetos no garantizan conservar el orden
    - En la mayor parte de implementaciones lo conservan

Maps

`Map` > `Object`

- API más completa y más conveniente
  - `.size`
  - `.has(...)`
  - `.clear(...)`
  - `...`

Maps

`Map` > `Object`

- Empiezan vacíos
  - Los objetos “vacíos” tienen varias propiedades predefinidas
    - `.constructor`, `.toString`, ….

Maps

`Map` > `Object`

- **Cualquier valor** se puede utilizar como clave
  - No está limitado a `String` o `Symbol`

Maps

```
const m = new Map();
m.set({ a: 1 }, 'value');
```

Maps

```
const m = new Map();
m.set({ a: 1 }, 'value');

console.log(m.get({ a: 1 })); // ???
```

Maps

```
const m = new Map();
const k = { a: 1 };
m.set(k, 'value');
```

Maps

```
const m = new Map();
const k = { a: 1 };
m.set(k, 'value');

console.log(m.get(k)); // ???
```

Maps

```
const a = new Map([['a', 1], ['b', 2]]);
const b = new Map([['a', 1], ['b', 2]]);

console.log(a === b); // ???
```

Ejercicio

- Implementa las operaciones:
  - `merge(A, B, C, ...)`
  - `equal(A, B)`
  - `deepEqual(A, B)`

Destructuring

Destructuring

- Una **sintaxis** que nos permite **“desmontar”** una estructura de datos
- Para hacer referencia a alguno de sus miembros
- Describiendo su “lugar” dentro de la estructura

```
const [a, b] = [1, 2]
```

```
const { x, y } = { x: 10, y: 20 }
```

Ejercicio

- Desestrucutura el objeto `{ uno: 1, dos: 2 }` en dos variables: **uno** y **dos**

Ejercicio

- Utiliza desestructuración para **intercambiar el valor** de las variables **a** y **b** *(sin crear ninguna otra variable!)*

```
let a = 1
let b = 2
// ???

console.log(a, b) // “2 1”
```

```
const { x: equis, y: igriega } = { x: 10, y: 20 }
```

```
const { x: { y } } = { x: { y: 10 } }
```

Ejercicio

- Desestructura el siguiente objeto en las variables **uno**, **dos**, **tres**, **cuatro** y **cinco**

```
{ uno: 1, lista: [2, 3], cuatro: 4, x: { cinco: 5 } }
```

Ejercicio

- Desestructura el siguiente objeto en las variables **a**, **b**, **c**, **d** y **e**

```
{ uno: 1, lista: [2, 3], cuatro: 4, x: { cinco: 5 } }
```

```
const [head] = [1, 2, 3]
```

```
const [, , tres] = [1, 2, 3]
```

Ejercicio

- Construye una **estructura de datos** que se pueda desestructurar con esta expresión:

```
const [{ lista: [ , { x: { y: dos } } ] }] = estructura
```

```
const [head, ...tail] = [1, 2, 3]
```

```
const [head, tail] = [ 1, 2, 3]

const [head, ...tail] = [1, 2]

const [head, ...tail] = [1]

const [head, , ...tail] = [1, 2, 3]
```

```
const lista1 = [1, 2, 3]
const [...lista2] = lista1

lista1 === lista2 // ??

let [a, b, c] = lista1
let [x, y, z] = lista2
a === x && b === y && c === z // ???
[a, b, c] === lista1 // ???

[a, b, c] = [x, y, z]
a === x && b === y && c === z // ???

[c, b, a] = [a, b, c]
a === x && b === y && c === z // ???
```

```
const lista = [1, 2, 3];

console.log(lista) // [1, 2, 3]

console.log(...lista) // 1 2 3

console.log(1, 2, 3) // 1 2 3
```

```
const lista1 = [1, 2]
const lista2 = [3, 4]
const a = [lista1, lista2] // ???
const b = [...lista1, ...lista2] // ???
```

```
const [a, b, c = 3] = [1, 2]
console.log(a, b, c) // 1 2 3
```

```
const { x: { y = 1 } = { y: 2 } } = { x: { y: 3 } }
const { x: { y = 1 } = { y: 2 } } = { x: { z: 3 } }
const { x: { y = 1 } = { y: 2 } } = { }

const [y = 10] = [2]
const [y = 10] = []
const [y = 10] = [1, 2]
const [y = 10] = [false]
const [y = 10] = [null]
const [y = 10] = [undefined]
```

```
function suma(a = 1, b = 1) {
  return a + b;
}

suma() // 2
suma(2) // 3
suma(2, 2) // 4
```

```
function someFunc({ x: equis, y: igriega = 10 }) {
  return equis + igriega;
}

someFunc({ x: 1, y: 10 }) // 11
someFunc({ x: 1 }) // 11
```

```
function sumaTodos(...args) {
  let total = 0;
  while (args.length) total += args.pop();
  return total;
}

sumaTodos(1)
sumaTodos(1, 1, 1, 1)
```