Fundamentos

Introducción

- La base del resto del curso
- Seguro que hay cosas que ya sabes
- Algunos conceptos de **ES6**
- ¿Qué es lo más fundamental que ofrece un lenguaje de programación?

Antes de empezar

Qué Necesitas

- Un editor de texto
- Una manera de ejecutar código javascript
  - Mi recomendación: instálate una versión reciente de node.js
  - Alternativa: una versión reciente de Chrome
- Clonar el repo del curso
  - `https://github.com/redradix/curso-javascript-pro`

Tipos de Datos

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
```

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

Tipos de Datos

¯_(ツ)_/¯

ES2015

ES2015

- Un nuevo tipo de datos:
  - Symbol
- Nuevas estructuras de datos:
  - Map
  - Set
  - Iteradores
  - Generadores
- Novedades para los tipos clásicos:
  - Template strings
  - Destructuring
  - Parámetros opcionales
  - Arrow functions

var, let y const

var, let y const

```
if (true) {
  var name = 'Homer';
  console.log(`Hola, ${name}`);
}
```

var, let y const

```
if (true) {
  var name = 'Homer';
  console.log(`Hola, ${name}`);
}
console.log(`Sigues ahí, ${name}??`);
```

var, let y const

```
console.log(`Sigues ahí, ${name}??`);
if (true) {
  var name = 'Homer';
  console.log(`Hola, ${name}`);
}
```

var, let y const

```
var name;
console.log(`Sigues ahí, ${name}??`);
if (true) {
  name = 'Homer';
  console.log(`Hola, ${name}`);
}
```

var, let y const

```
function greet() {
  console.log(`Sigues ahí, ${name}??`);
  if (true) {
    var name = 'Homer';
    console.log(`Hola, ${name}`);
  }
}
```

var, let y const

`let`

- declara una variable
- ámbito local al bloque en el que se declara
- no hoisting

var, let y const

```
if (true) {
  let name = 'Homer';
  console.log(`Hola, ${name}`);
}
console.log(`Sigues ahí, ${name}??`);
```

var, let y const

```
ReferenceError: name is not defined
```

var, let y const

```
console.log(`Sigues ahí, ${name}??`);
if (true) {
  let name = 'Homer';
  console.log(`Hola, ${name}`);
}
```

var, let y const

```
ReferenceError: name is not defined
```

var, let y const

`let`

- Comportamiento más predecible que **`var`**
- Ámbito mejor definido
- **`let > var`**

var, let y const

`const`

- declara una **constante**
- ámbito local al bloque en el que se declara
- no hoisting

var, let y const

```
if (true) {
  const name = 'Homer';
  console.log(`Hola, ${name}`);
}
console.log(`Sigues ahí, ${name}??`);
```

var, let y const

```
ReferenceError: name is not defined
```

var, let y const

```
const name = 'Homer';
name = 'Fry';
```

var, let y const

```
TypeError: Assignment to constant variable.
```

var, let y const

Deja de usar **`var`**

- Usa **`const`** siempre que sea posible
  - Claridad, expresividad, seguridad
- Usa **`let`** para todos los demás casos
  - Comportamiento más evidente

Destructuring, rest y splice

Destructuring

Permite *desempaquetar* valores de array o de objetos en diferentes variables

- se pueden anidar
- se pueden aplicar en las asignaciones implícitas
  - parámetros de función
  - for

Destructuring

```
let [a, b] = [1, 2];

console.log(a); // 1
console.log(b); // 2
```

Destructuring

```
[b, a] = [a, b];

console.log(a); // 2
console.log(b); // 1
```

Destructuring

```
const { y, x } = { x: 100, y: 200 };

console.log(x); // 100
console.log(y); // 200
```

Destructuring

```
let [[a, b], c] = [[1, 2], 3];

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

Destructuring

```
let { x: { y: { z }}} = { x: { y: { z: 1 } } };

console.log(z); // 1
```

Destructuring

```
let { x: { y: { z }}} = { x: { y: { z: 1 } } };

console.log(z); // 1
console.log(x); // Error: x is not defined!
```

Destructuring

```
const [a, , b] = [1, 2, 3];

console.log(a); // 1
console.log(b); // b
```

Destructuring

```
const { x: equis, y: igriega } = { x: 1, y: 2 };

console.log(equis); // 1
console.log(igriega); // 2
```

redradix

Destructuring

```
let [a = 1, b, c = 0] = [2, 3];

console.log(a); // 2
console.log(b); // 3
console.log(c); // 0
```

Destructuring

```
let { x = 1, y = 2 } = { z: 1, y: 3 };

console.log(x); // 1
console.log(y); // 3
```

Destructuring

```
let { x: { z } = { z: 10 } } = { z: 25 }

console.log(z); // ???
```

Destructuring

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

console.log(head); // 1
console.log(tail); // [2, 3]
```

Destructuring

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

console.log(list); // [1, 2, 3]
```

Destructuring

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

console.log(list); // [1, 2, 3, 2, 3]
```

Destructuring

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

const params = [1, 2];
sum(...params); // 3
```

Destructuring

```
function sum(a = 1, b = 2) {
  return a + b;
}

sum(10); // 12
```

Destructuring

```
function log(...rest) {
  console.log(...rest);
}

log('very', 'useful', 'function');
```

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 wat = (a) => a ? 1 : 2;
```

Arrow functions

```
const wat = ((a) => a) ? 1 : 2;
```

Arrow functions

```
const wat = (a) => (a ? 1 : 2);
```

Arrow functions

```
const wat = (a) => console.log(a), 1;
```

Arrow functions

```
const wat = (a) => (console.log(a), 1);
```

Arrow functions

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

Arrow functions

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

Clausuras

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

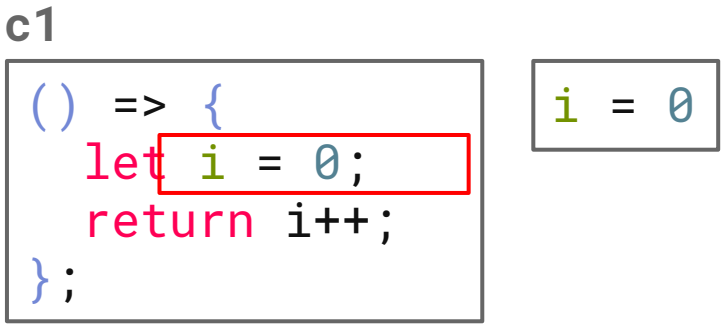
```
const c1 = counter();
c1();
```

c1

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

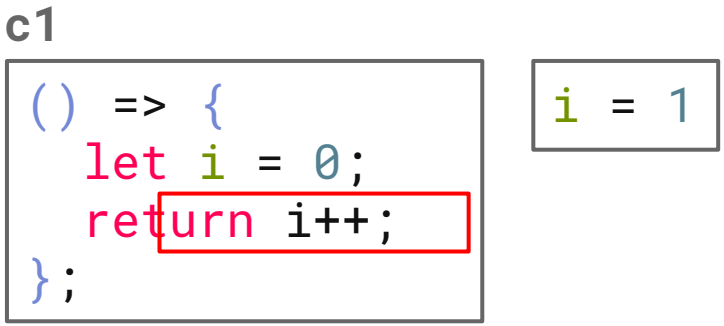
Clausuras

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



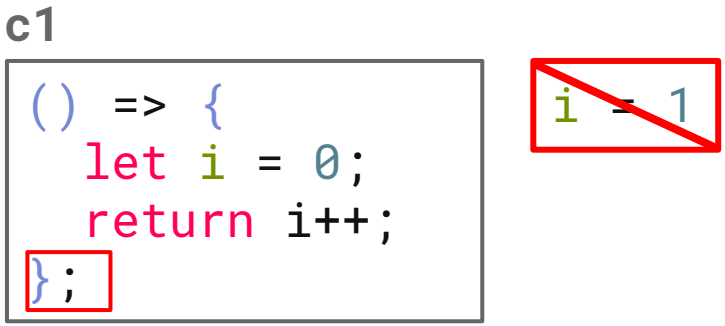
Clausuras

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



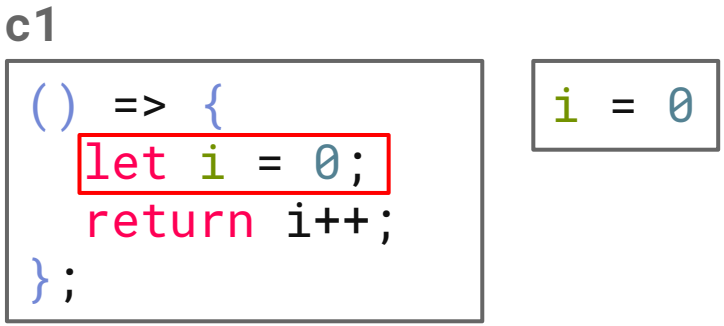
Clausuras

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



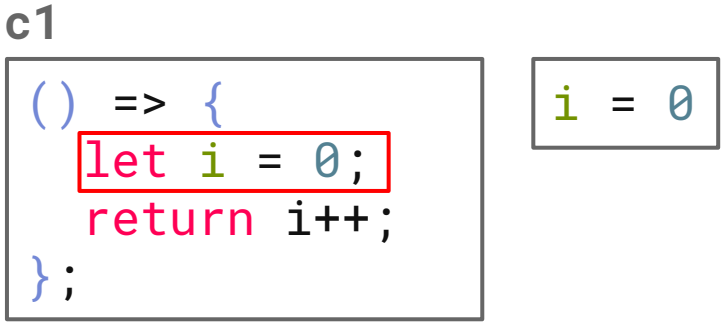
Clausuras

```
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

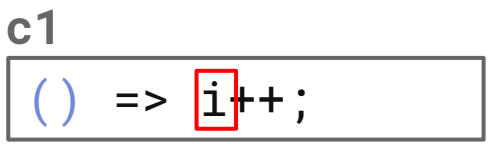
```
const c1 = counter();
console.log(c1()); // 0
console.log(c1());
```

Clausuras

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

Clausuras

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

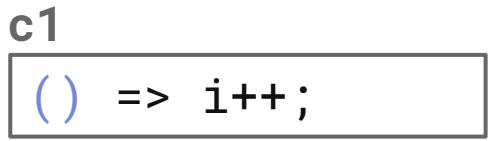


Clausuras

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

Clausuras

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



Clausuras

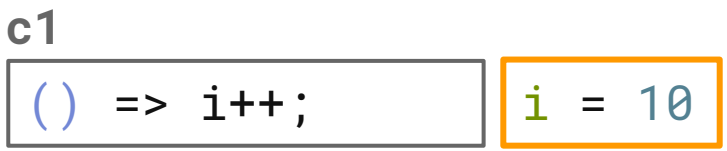
```
const c1 = counter();
let i = 10;
c1();              // ???
console.log(i); // ???
```

c1

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

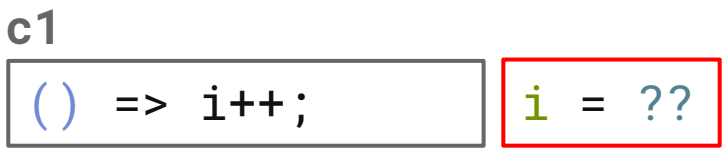
Clausuras

```
const c1 = counter();
let i = 10;
c1();              // 10
console.log(i); // 11
```



Clausuras

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

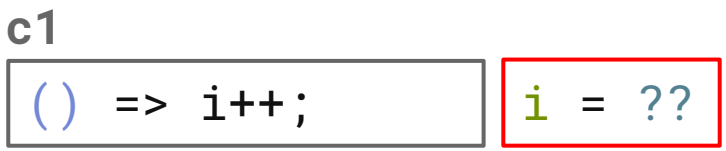


Clausuras

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

Clausuras

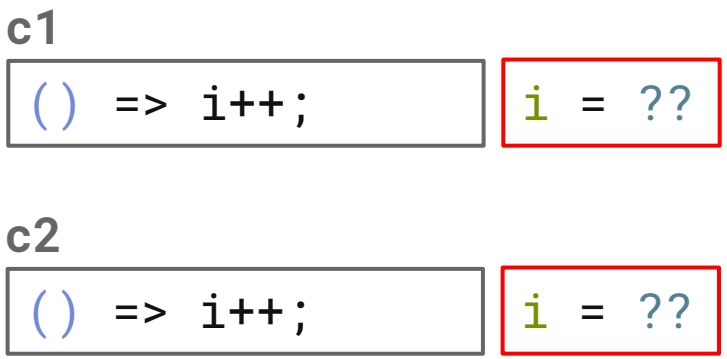
```
const c1 = counter();
c1(); // 0
c1(); // 1
```



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*

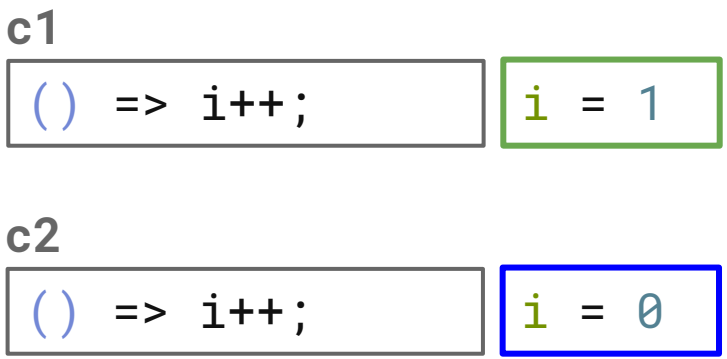
Clausuras

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

Clausuras

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

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



redradix

Symbols

Symbols

- Primer tipo de datos nuevo desde 1997
- Función muy especializada
- Similar al 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

Symbols

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

Symbols

- No se convierte automáticamente a `string`
- No se puede añadir propiedades a un símbolo
  - (en modo estricto)

Symbols

```
function test() {
  'use strict';
  const a = Symbol();
  a.prop = 1;
}

test();
```

Symbols

```
TypeError: Cannot create property 'prop' on symbol
'Symbol()'
```

Symbols

```
console.log('Symbol:' + Symbol());
```

Symbols

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

Symbols

```
console.log('Symbol:' + String(Symbol()));

// Symbol:Symbol()
```

Symbols

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

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

Symbols

3 maneras de definir símbolos:

- `Symbol(`*`[description]`*`)`
- `Symbol.for(`*`id`*`)`
- Símbolos predefinidos

Symbols

`Symbol([description])`

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

Symbols

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

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

Symbols

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

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

redradix

Symbols

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

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

Symbols

```
function test() {
  'use strict';
  const a = Symbol('a');
  a.prop = 1;
}

test();
```

Symbols

```
TypeError: Cannot create property 'prop' on symbol
'Symbol(a)'
```

Symbols

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

a1 === a2;
```

Symbols

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

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

Symbols

3 maneras de definir símbolos:

- `Symbol(`*`[description]`*`)`
- `Symbol.for(`*`id`*`)`
- Símbolos predefinidos

Symbols

`Symbol.for(`*`id`*`)`

- Accede a un registro global de símbolos
- Devuelve el mismo símbolo para un mismo *id*

Symbols

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

a1 === a2; // true
```

Symbols

```
const a = Symbol.for('a');

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

Symbols

3 maneras de definir símbolos:

- `Symbol(`*`[description]`*`)`
- `Symbol.for(`*`id`*`)`
- Símbolos predefinidos

Symbols

El estándar especifica algunos símbolos predefinidos

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

Symbols

Pero los símbolos... ***¿Para qué sirven?***

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...

- Acceder a propiedades
- Que sólo son accesibles con ese símbolo
- Sin la referencia al símbolo son invisibles

Symbols

Aplicaciones:

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

Iterators

Iterators

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

Iterators

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

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]);

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

Iterators

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

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

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 un objeto que js sabe como recorrer
- **for ... of**
- 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:**
  - `[Symbol.iterator]`
  - devuelve un iterador

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* 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('Va un uno!');
  yield 1;
  console.log('Va un dos!');
  yield 2;
  console.log('Va un tres!');
  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
```

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

Pero... ***todo esto se puede hacer con objetos***

de toda la vida

Maps

¯_(ツ)_/¯

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); // ???
```

Maps: Ejercicio

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

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);
}
```

Sets: Ejercicio

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

Sets: 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' }
```

OBJECT

redradix

I

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 = new Object();
```

Object

```
const obj = {};
obj.a = 1;
obj['b'] = 2;
const k = 'c';
obj[k] = 3;
obj[([][[]]+'')[2]] = 4;
delete obj.a;
```

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; // ???
```

II

redradix

Object

- `Object.assign(target, ...sources)`
- `Object.keys(obj)`
- `Object.values(obj)`

Object

```
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 };
```

Object

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

const x = Object.assign({}, a, b, c);
Object.assign(x, { d: 4 });
console.log(x); // { a: 1, b: 2, c: 3, d: 4 }
```

Object

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

const ks = Object.keys(obj);
const vs = Object.values(obj);

console.log(ks); // [ 'a', 'b' ]
console.log(vs); // [ 1, 2 ]
```

Object

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

const ks = Object.keys(obj);
const vs = Object.values(obj);

console.log(ks); // [ 'a', 'b' ]
console.log(vs); // [ 1, 2 ]

console.log(obj.toString) // [ Function: toString]
```

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

`Object.defineProperty(obj, prop, desc)`

- Configura una propiedad del objeto
  - Existente o no existente
  - Control absoluto sobre el comportamiento
  - `Object.defineProperties(...)` para multiples props

Object

Descriptor de propiedad:

- value
- enumerable
- configurable
- writable

Object

```
const obj = {};

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

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

Object

```
const obj = {};

Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
});

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

Object

```
const obj = {};

Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
});

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

Object

```
const obj = {};

Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
});

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

Object

```
const obj = {};

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

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

Object

```
const obj = {};

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

console.log(Object.keys(obj)); // [ 'b', 'c' ]
```

Object

```
const obj = {};

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

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

Object

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

Object

```
const obj = {};

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

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

redradix

Object

```
const obj = {};

Object.defineProperty(obj, 'd', {
  value: 1,
  enumerable: true,
  writable: false
});

obj.d = 2;
console.log(obj.d); // 1
```

redradix

Object

Descriptor de propiedad:

- value (*undefined*)
- enumerable (*false*)
- configurable (*false*)
- writable (*false*)

Object

- 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); // ???
```

III

redradix

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

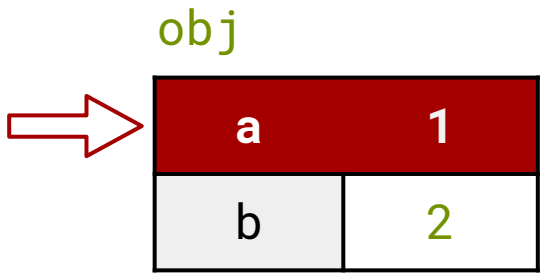
```
const obj = { a: 1, b: 2 };
```

obj

a	1
b	2

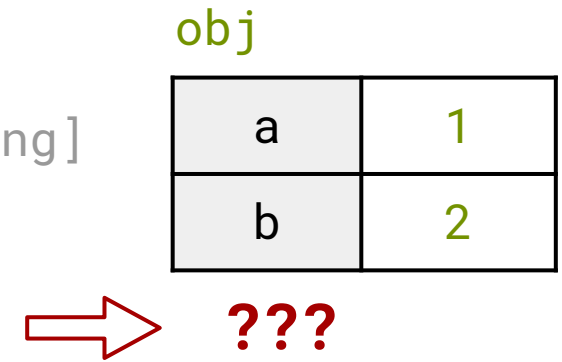
Object

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



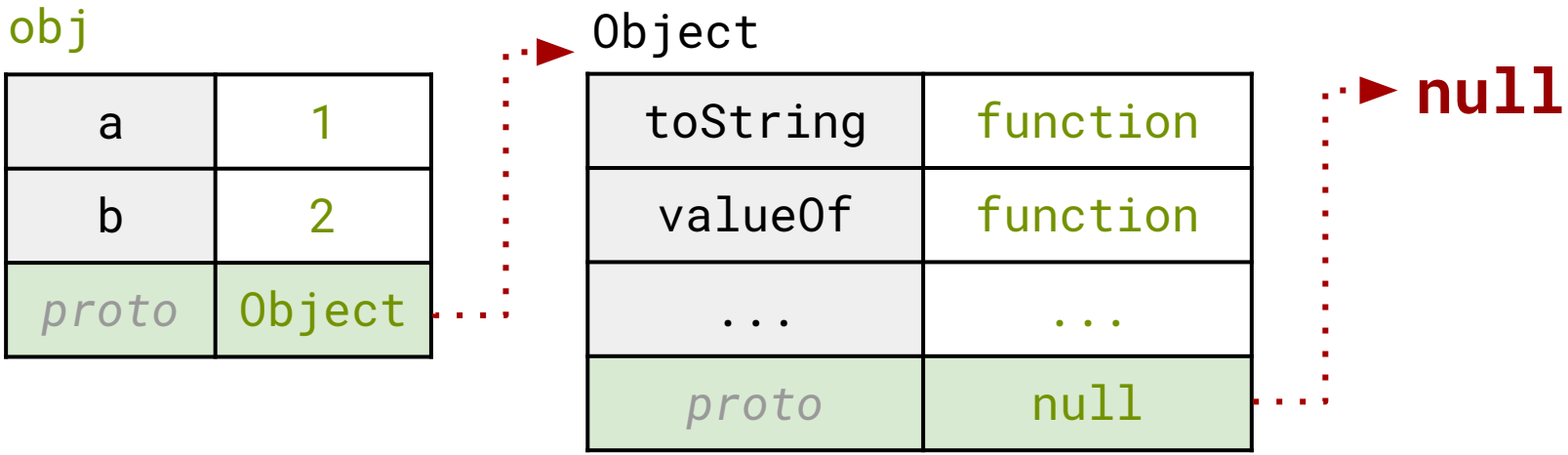
Object

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



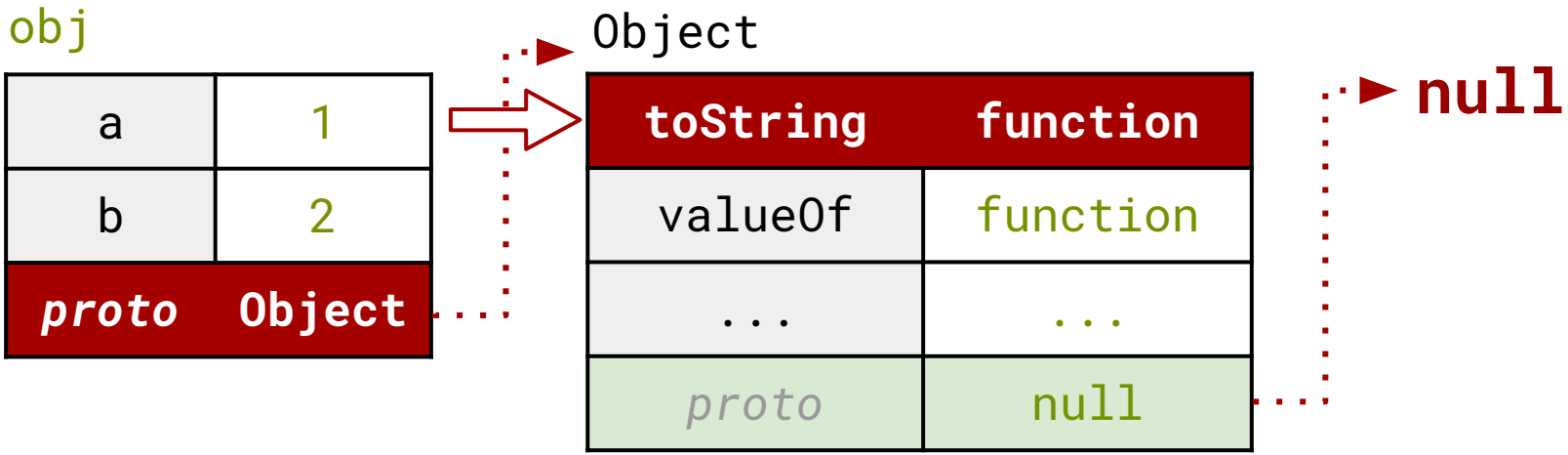
Object

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



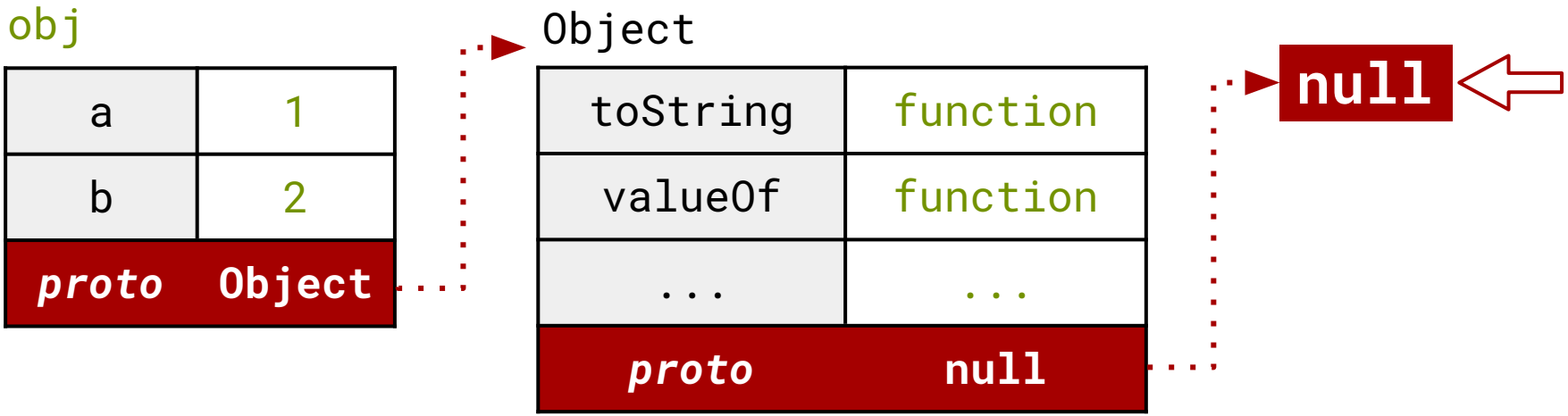
Object

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



Object

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

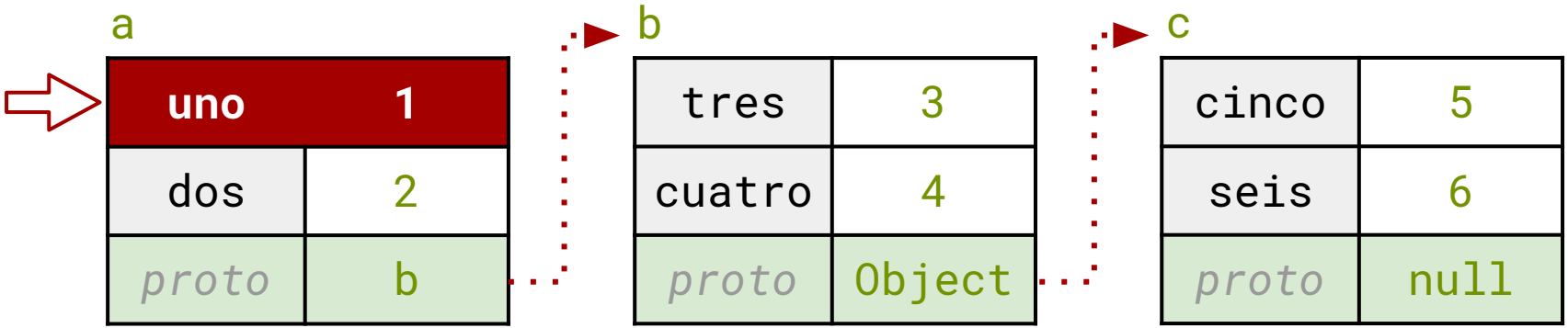


Object

- 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: Prototipos

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

Ejercicio: Prototipos

- ¿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; // ???
```

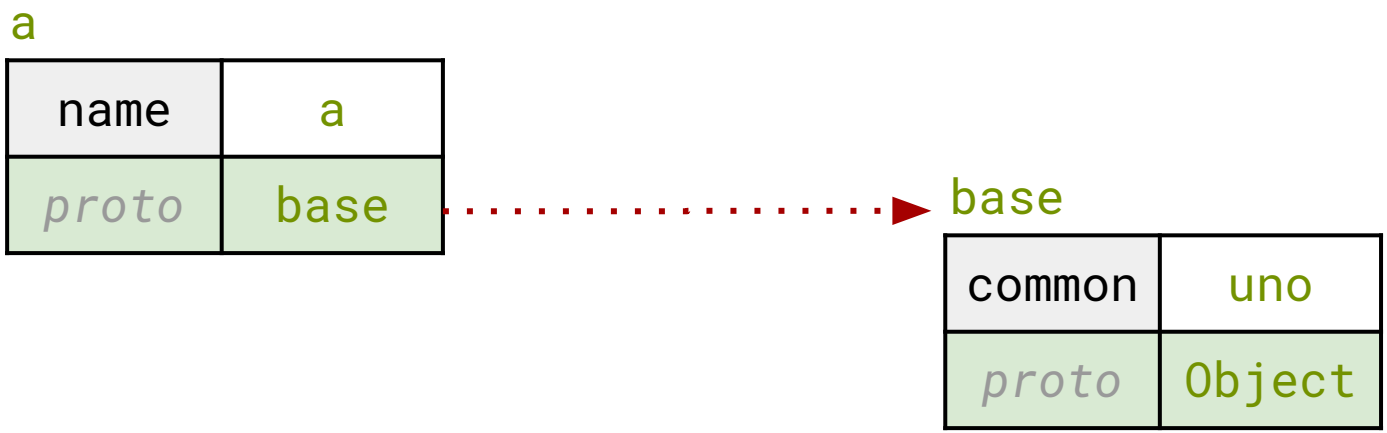
Object

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

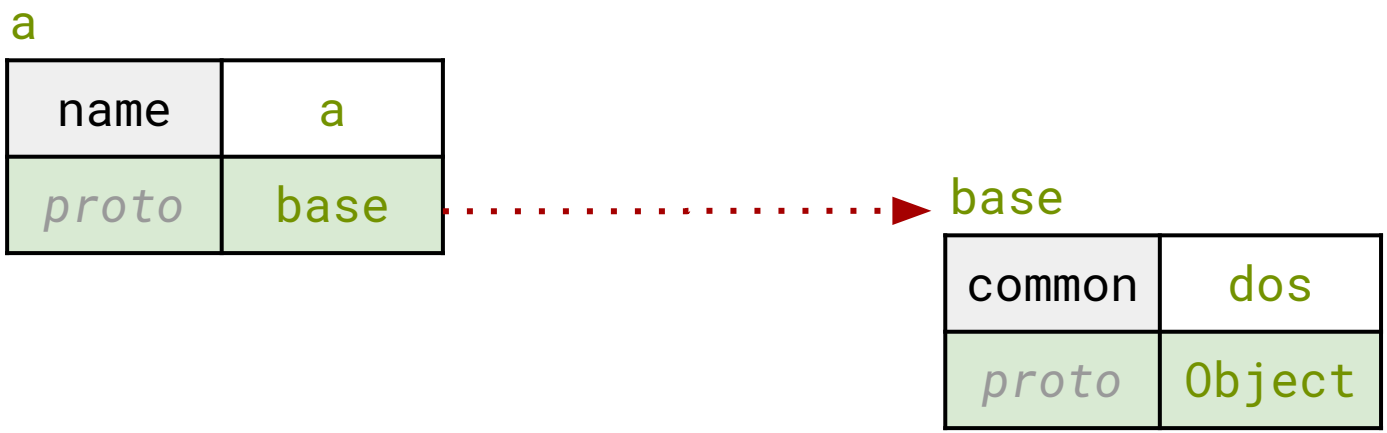
Object

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

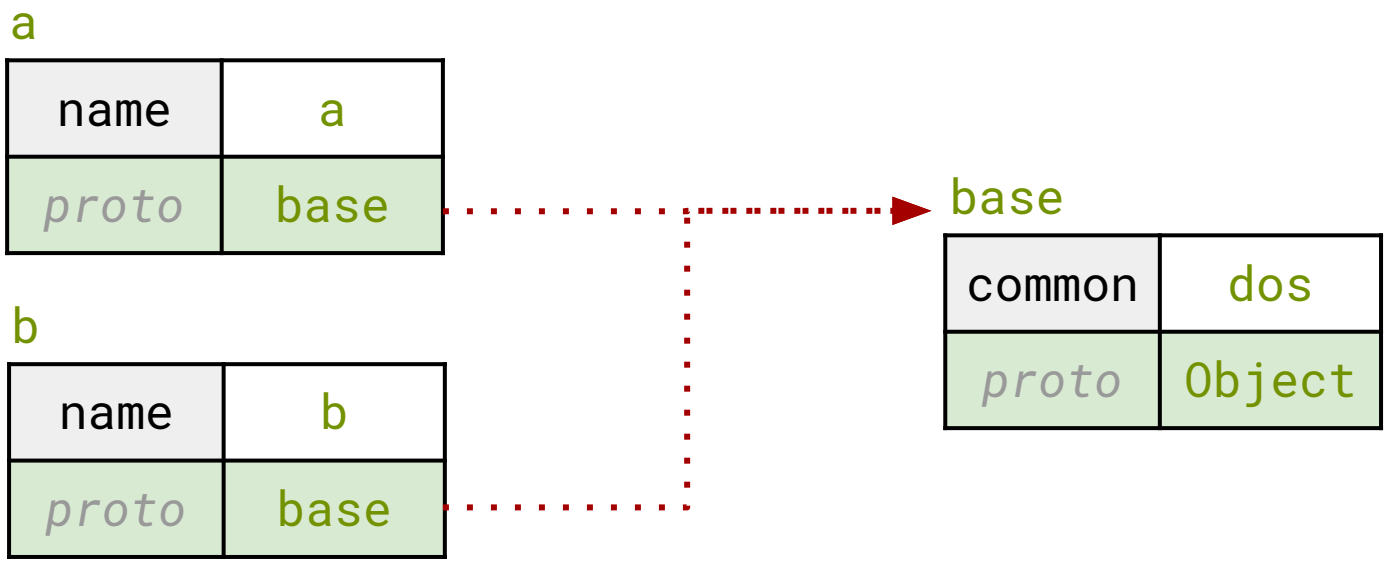
Object



Object



Object



Object

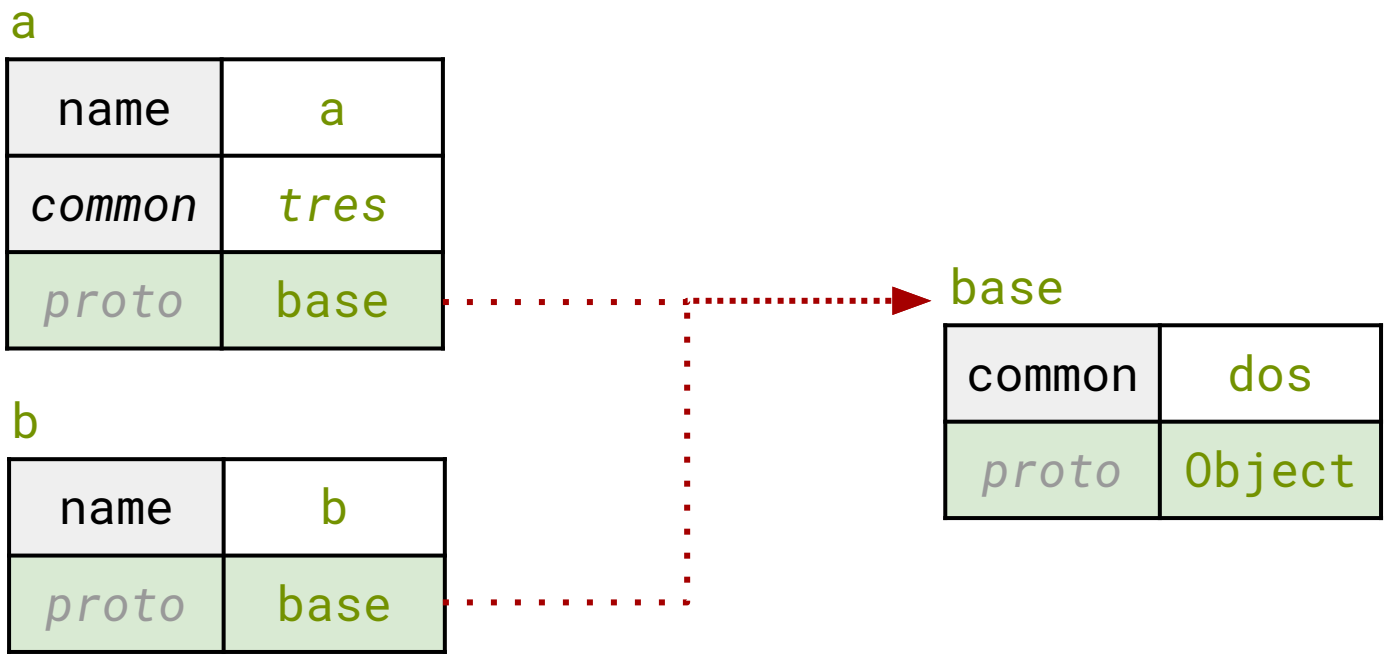
```
a.common = 'tres';
b.common; // ???
```

Object

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

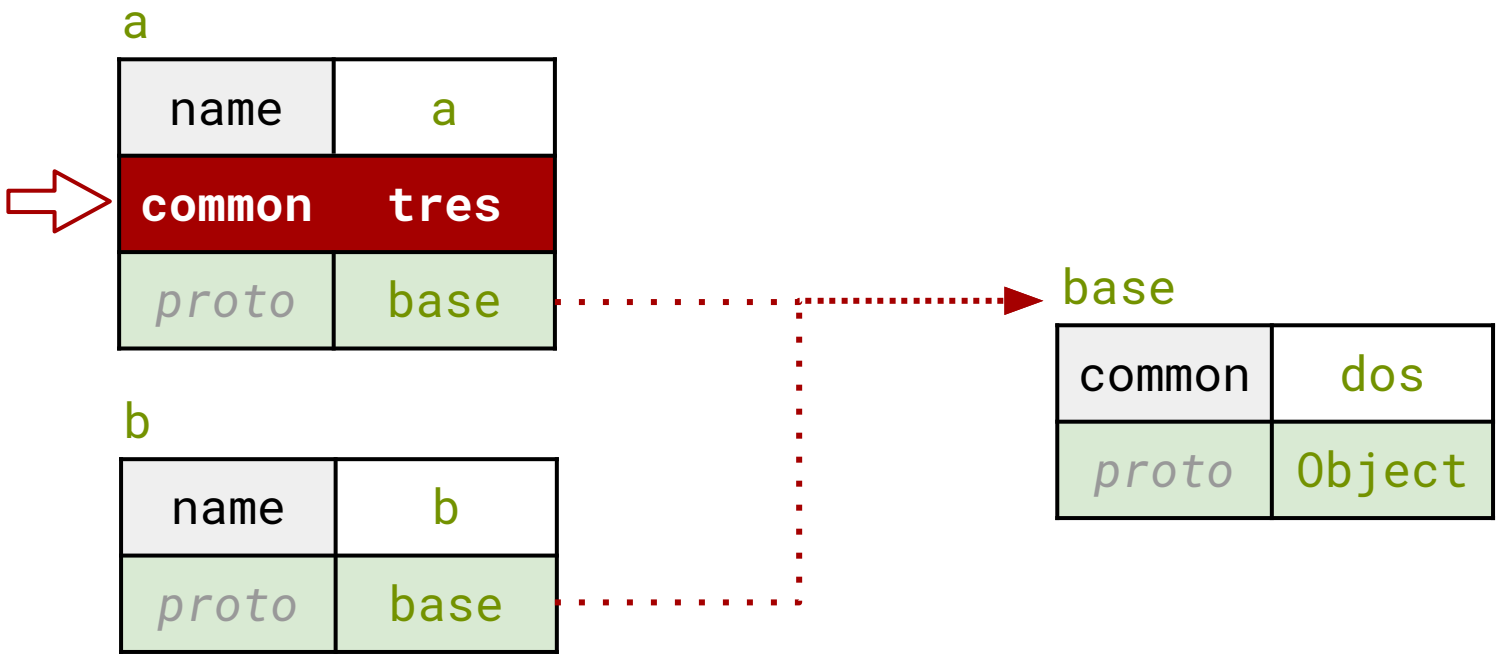
Object

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

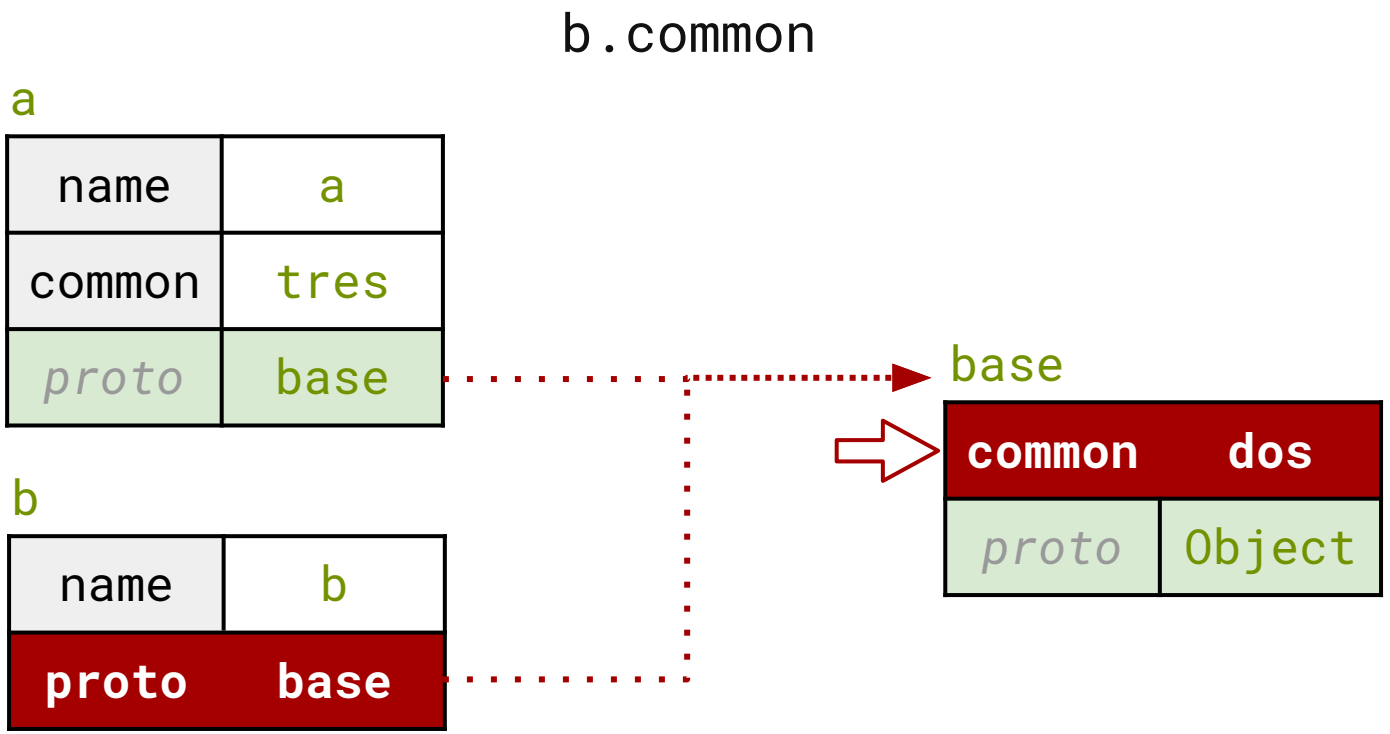


Object

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

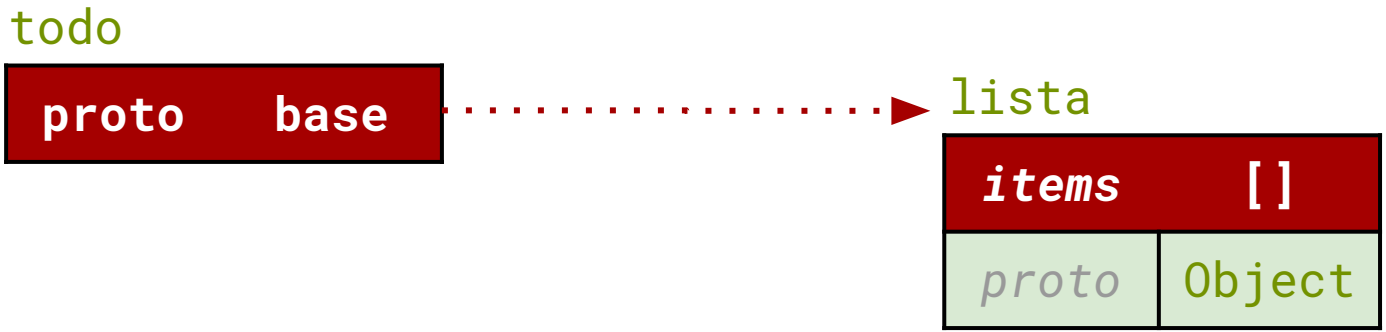
```
const todo = Object.create(lista);
```

Object

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

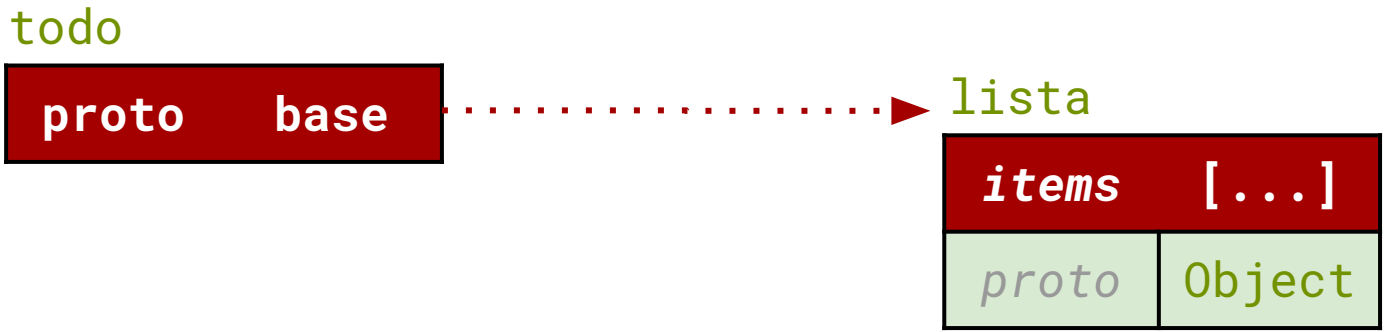
Object

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



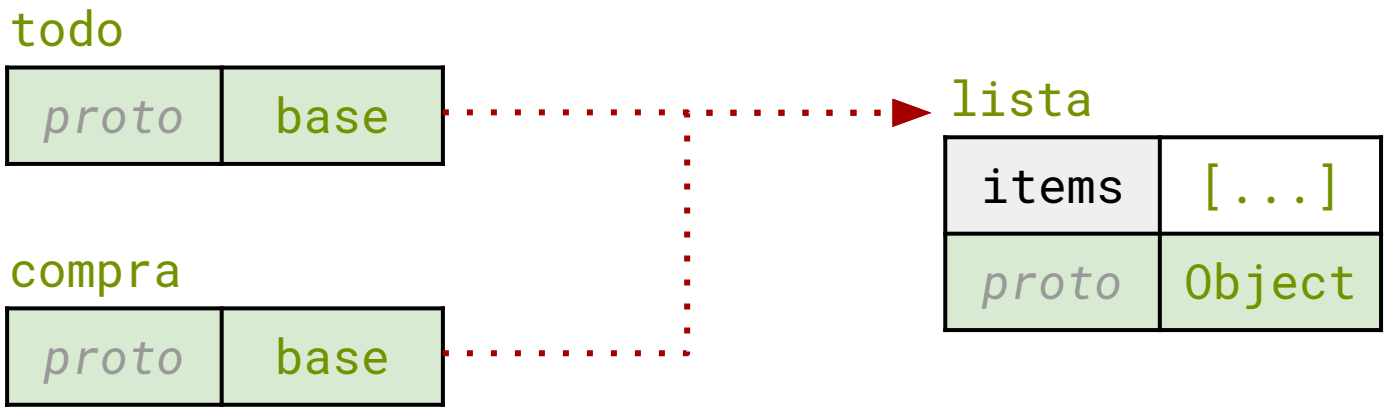
Object

```
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; // ???
```