CLASS(E)

Módulo 4. Estructuras de datos II

Objetos

Objetos

JavaScript ofrece 1 tipo de dato compuesto:

- Object

¿Y los arrays?

Objetos

En JavaScript los arrays son objetos.

```
typeof [1, 2] // 'object'
```

Object

Un conjunto de propiedades clave/valor.

- clave: string o symbol
- valor: cualquier tipo de valor

Puede heredar propiedades de otro objeto.

Manejado por referencia.

Casi todo en JavaScript son Objects

Creación de objetos

```
const obj = {}

const obj2 = { prop: 1}

const obj3 = { "example-prop": 1}
```

Objetos

[] nos permite crear claves dinámicas.

```
const obj = { ["a" + "b"]: 1 }
console.log(obj) // ?
```

Objetos

[] nos permite crear claves dinámicas.

```
const obj = { ["a" + "b"]: 1 }
console.log(obj) // {"ab": 1}
```

Objetos

[] nos permite crear claves dinámicas.

```
const key = "lorem"
const obj = { [key]: 1 }
console.log(obj) // ?
```

Objetos

[] nos permite crear claves dinámicas.

```
const key = "lorem"
const obj = { [key]: 1 }
console.log(obj) // {"lorem": 1}
```

Igualdad en objetos

¿Qué imprime el siguiente código?

```
const obj1 = {"Homer": "Simpson"}
const obj2 = {"Homer": "Simpson"}

console.log(obj1 === obj2) // ?
```

Igualdad en objetos

¡Falso!

```
const obj1 = {"Homer": "Simpson"}
const obj2 = {"Homer": "Simpson"}

console.log(obj1 === obj2) // False
```

Igualdad en objetos

Los objetos se consideran iguales si comparten referencia.

```
const obj1 = {"Homer": "Simpson"}
const obj2 = obj1
const obj3 = obj1

console.log(obj2 === obj3) // True
```

Objetos

Teniendo eso en cuenta...

```
const obj1 = { count: 1}
const obj2 = obj1
obj2.count = 2

console.log(obj1 === obj2) // ?
```

Objetos

Teniendo eso en cuenta...

```
const obj1 = { count: 1}
const obj2 = obj1
obj2.count = 2

console.log(obj1 === obj2) // True
```

A todo esto...

Objetos

¿Por qué obj2.count=2 no genera error siendo obj2 una constante?

```
const obj1 = { count: 1}
const obj2 = obj1
obj2.count = 2
```

Recorrer objetos

Recorrer objetos

Disponemos de tres métodos básicos para iterar objetos.

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

let result = Object.keys(obj)
// [‘a’, ‘b’, ‘c’]

let result = Object.values(obj)
// [1, 2, 3]

let result = Object.entries(obj)
// [ [ 'a', 1 ], [ 'b', 2 ], [ 'c', 3 ] ]
```

Ejercicio objetos

Implementa la función mapKeys que transforma todas las claves de un objeto.

```
const obj = {a: 1, b: 2, c: 3}
const result = mapKeys(obj, key =>
key.toUpperCase())
console.log(result) // {A: 1, B: 2, C: 3}
```

Ejercicio objetos II

Implementa la función mapValues que transforma todos los valores de un objeto.

```
const obj = {a: 1, b: 2, c: 3}
const result = mapValues(obj, x => x * 2)
console.log(result) // {a: 2, b: 4, c: 6}
```

Object.assign

Object.assign

- Nos permite fusionar objetos
- Asignando las propiedades de un objeto a otro
- De derecha a izquierda

Object.assign

- Nos permite fusionar objetos
- Asignando 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) // ?
```

Object.assign

- Nos permite fusionar objetos
- Asignando las propiedades de un objeto a otro
- De derecha a izquierda

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

Object.assign(a, b, c)

console.log(a) // ?
console.log(b) // ?
console.log(c) // ?
```

Ejercicio objetos III

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

Ejercicio objetos IV

Crea una función clone que reciba un objeto y devuelva una copia (referencia distinta).

Objetos

Podemos añadir valores de cualquier tipo a un Objeto.

```
let  div = {
    "colors": ["red", "blue"], // <-- Arrays
    "dimensions": {"width": 400, "height":300}, // <-- Objetos
    "remove": function(){ // <-- Funciones
        console.log("div has been removed")
    }
}
```

Ejercicio objetos V 🔥

```
{a: 1, b: {c: 2, d: 5, e: {f: 9, g: 6}}}
```

Crea una función que reciba un objeto como este y sume todos los números.

El resultado debería ser 23.

Ejercicio objetos VI 🔥

```
{a: 1, b: {c: 2, d: 5, e: {f: 9, g: 6}}}
```

Crea una función traverse que aplica una función a todas las hojas de este tipo de objetos recursivos.

Consideramos hojas cualquier valor que no sea un objeto.

Ejercicio objetos VII 🔥🔥

Crea una función `cloneDeep`:

- Versión recursiva de `clone`
- Ningún subobjeto mantiene la referencia anterior
  - ¡Incluyendo arrays!

Ejercicio objetos VIII 🔥🔥🔥

Crea una función `mergeDeep`:

- Versión recursiva de `Object.assign`
- Ningún subobjeto mantiene la referencia anterior
- Casos de prueba en las siguientes diapositivas →

```
const obj1 = { a: { b: { c: 1 } } }
const obj2 = { a: { b: { d: 2 } } }

const result = mergeDeep({}, obj1, obj2)
console.log(result.a.b) // { c: 1, d: 2 }
```

Casos de prueba para testear merge

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

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

Casos de prueba para testear merge

```
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)
// { a: { b: { c: 1, d: 2 }, c: 'hey' }, b: 2, c: 4, x: 3 }

console.log(u1) // igual que x
console.log(u2) // no cambia
console.log(u3) // no cambia
```

Destructuring

Los objetos también se pueden desestructurar.

```
const { x, y } = { x: 10, y: 20 }
console.log(x) // 10
console.log(y) // 20
```

Ejercicio destructuring

Desestructura el objeto { uno: 1, dos: 2 } en dos variables: uno y dos.

Ejercicio destructuring II

Utiliza la desestructuración para intercambiar el valor de las variables a y b.

- No crees una tercera variable.

```
let a = 1
let b = 2
// ?
console.log(a, b) // 2 1
```

Destructuring

Podemos cambiar el nombre de las variables al desestructurarlas.

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

console.log(equis) // 10
console.log(ye) // 20
```

Destructuring

Podemos desestructurar de forma anidada.

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

// ?
```

Ejercicio destructuring III

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

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

Ejercicio destructuring IV

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

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

Destructuring

Podemos aplicar la desestructuración en los parámetros de una función.

```
function func({ x, y = 10}) {
  return x + y;
}

func({ x: 1, y: 20 }) // 21
func({ x: 1 }) // 11
```

Destructuring

Podemos aplicar la desestructuración en los parámetros de una función.

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

func({ x: 1, y: 20 }) // 21
func({ x: 1 }) // 11
```

Spread

El operador spread también funciona con Objetos.

```
const obj = { a: 1 }

const obj2 = { b: 2, c: 3 }

const obj3 = { ...obj, ...obj2 }
// { a: 1, b: 2, c: 3}
```

Spread

Muy útil para hacer copias superficiales de objetos.

```
const obj = { lorem: "ipsum" }
const shallowCopy = { ...obj }
```

Spread

Muy útil para hacer copias superficiales de objetos.

```
const obj = {a: 1, b: 2}
const obj2 = {...obj, a: 2} // {a: 2, b: 2}
```

Contexto

Scope vs. Contexto

El scope hace referencia a la visibilidad de las variables.

El contexto hace referencia al objeto al que pertenece una función.

Accedemos al contexto mediante el término this.

Contexto

El contexto hace referencia al objeto al que pertenece una función.

```
let obj = {
    prop1: "aaa",
    prop2: "bbb",
    action: function(){
        console.log(this)
    }
}

obj.action() // ?
```

Contexto

El contexto hace referencia al objeto al que pertenece una función

```
let obj = {
    prop1: "aaa",
    prop2: "bbb",
    action: function(){
        console.log(this)
    }
}

obj.action()
/* {
     prop1: 'aaa',
     prop2: 'bbb',
     action: [Function: action]
} */
```

Contexto

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: function(){
            console.log(this)
        }
    },
    action: function(){
        console.log(this)
    }
}

obj.obj2.action() // ?
```

Contexto

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: function(){
            console.log(this)
        }
    },
    action: function(){
        console.log(this)
    }
}

obj.obj2.action()
// { name: 'obj2', action: [Function:
action] }

obj.action() // ?
```

Contexto

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: function(){
            console.log(this)
        }
    },
    action: function(){
        console.log(this)
    }
}

obj.obj2.action()
// { name: 'obj2', action: [Function: action] }

obj.action()
/* {
      name: 'obj',
      obj2: {
            name: 'obj2',
            action: [Function: action]
            },
      action: [Function: action]
} */
```

¿Qué es this en el scope global?

Vamos a comprobarlo

Contexto en arrow functions

Contexto en arrow functions

El this en una arrow function no hace referencia al objeto al que pertenece.

El this en una arrow function es el mismo dentro que fuera.

Contexto en arrow functions

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: () => {
            console.log(this)
        }
    },
    action: () => {
        console.log(this)
    }
}

obj.action() // ?

obj.obj2.action()
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: () => {
            console.log(this)
        }
    },
    action: () => {
        console.log(this)
    }
}

obj.action() // window/global

obj.obj2.action() // ?
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: () => {
            console.log(this)
        }
    },
    action: () => {
        console.log(this)
    }
}

obj.action() // window/global


obj.obj2.action() // window/global
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    action: function(){
        let name = "function"
        let obj2 = {
            name: "obj2",
            action: () => {
                console.log(this)
            }
        }
        obj2.action()
    }
}

obj.action() // ?
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    action: function(){
        let name = "function"
        let obj2 = {
            name: "obj2",
            action: () => {
                console.log(this)
            }
        }
        obj2.action()
    }
}

// { name: 'obj', action: [Function: action] }
obj.action()
```

Alteración de contexto

Alteración de contexto

¿Qué imprime esto?

```
const Peter = { name: "Peter"}
const obj = {
    name: "Homer"
    greet: function(surname) {
        console.log(`Hola, soy ${this.name} ${surname}`)
    }
}

obj.greet("Simpson") // ?
```

Alteración de contexto

```
const Peter = { name: "Peter"}
const obj = {
    name: "Homer"
    greet: function(surname) {
        console.log(`Hola, soy ${this.name} ${surname}`)
    }
}

obj.greet("Simpson") // Hola, soy Homer Simpson
```

Alteración de contexto

`call` nos permite asignar un contexto a la llamada de una función.

```
const Peter = { name: "Peter"}
const obj = {
    name: "Homer"
    greet: function(surname, surname2) {
        console.log(`Hola, soy ${this.name} ${surname} ${surname2}`)
    }
}

obj.greet.call(Peter, "J.", "Simpson") // ?
```

Alteración de contexto

`call` nos permite asignar un contexto a la llamada de una función.

```
const Peter = { name: "Peter"}
const obj = {
    name: "Homer"
    greet: function(surname, surname2) {
        console.log(`Hola, soy ${this.name} ${surname} ${surname2}`)
    }
}

obj.greet.call(Peter, "J.", "Simpson") // Hola, soy Peter J. Simpson
```

Alteración de contexto

`apply` funciona igual que `call`, pero recibe los argumentos en forma de array.

```
const Peter = { name: "Peter"}
const obj = {
    name: "Homer"
    greet: function(surname, surname2) {
        console.log(`Hola, soy ${this.name} ${surname} ${surname2}`)
    }
}

obj.greet.apply(Peter, ["J.", "Simpson"]) // Hola, soy Peter J. Simpson
```

Ejercicio contexto

Arregla este código sin modificar func ni obj

```
function func() {
    console.log(this.num) // Debería imprimir 10
}

let obj = {
    callFun : func
}

obj.callFun()
```

Ejercicio contexto II

Crea una función `bind` que reciba los parámetros `context` y `func`

- Debe devolver una versión de func que se ejecute usando el contexto context
- Utiliza call o apply

Bind

`bind` existe de forma nativa en JavaScript.

```
const Peter = { name: "Peter"}
const obj = {
    name: "Homer"
    greet: function(surname, surname2) {
        console.log(`Hola, soy ${this.name} ${surname} ${surname2}`)
    }
}

const bound = obj.greet.bind(Peter)
bound("J.", "Simpson") // Hola, soy Peter J. Simpson
```

Object.defineProperty

defineProperty

`Object.defineProperty` nos permite `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

defineProperty

Recibe tres parámetros:

- El objeto
- El nombre de la propiedad
- El descriptor de propiedad

```
const obj = {}

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

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

defineProperties

El método `Object.defineProperties` nos permite modificar varias propiedades a la vez

```
const obj = {}

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

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

defineProperty

¿Qué imprime esto?

```
const obj = {}

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

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

Enumerables

Las propiedades b y c no se ven porque no son enumerables.

Valor no enumerable:

- el valor está ahí si accedes directamente a él
- pero no aparece en la lista de propiedades del objeto

Enumerables

Podemos hacer que las variables sean enumerables.

```
const obj = {}

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

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

Enumerables

Probad a ejecutar el siguiente código:

```
const obj = {}

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

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

TypeError: Cannot redefine property: a

Configurable

Para poder redefinir una propiedad con defineProperty, debemos hacerla configurable.

```
const obj = {}

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

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

defineProperty

Valores por defecto:

- value → undefined
- enumerable → false
- configurable → false
- writable → false

defineProperty

Este nivel de control sobre las propiedades de los objetos nos permite:

- ocultar propiedades que no queremos exponer
- crear propiedades que no pueden ser sobreescritas
- que no se puedan sobrescribir las dos definiciones anteriores (usando no configurable)

Getters y setters

defineProperty

El descriptor de propiedad también puede especificar:

- get
- set

Get

`get` define una función que se invoca al acceder a una propiedad.

```
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
```

Get

¿Qué imprime este código?

```
const obj = {};

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

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

Set

`set` es una función que se ejecuta al escribir el valor de una propiedad.

```
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;
  }
});

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

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

Get y Set

¿Qué pasa si copio un valor con getters y setters?

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

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

Sólo se copia el valor calculado

Get y Set

Podemos definir getters y setters directamente sin defineProperty.

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

Ejercicio getters y setters

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

Ejercicio getters y setters II

Crea un objeto con una propiedad `value`

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

Prototipos

Object

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

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

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

Object

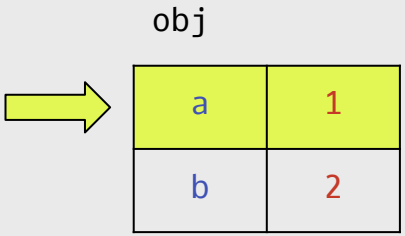
```
const obj = { a: 1, b: 2 };
```

obj

a	1
b	2

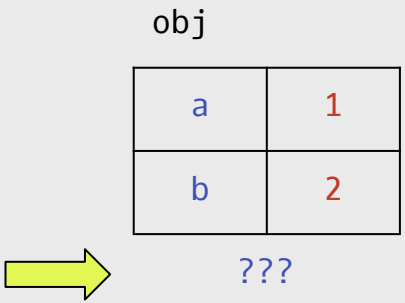
Object

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



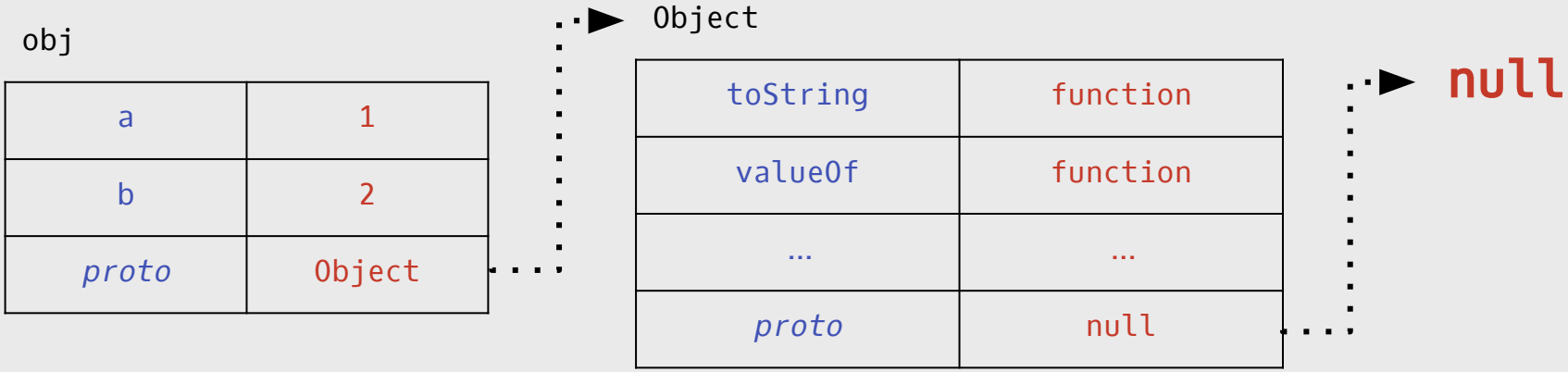
Object

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



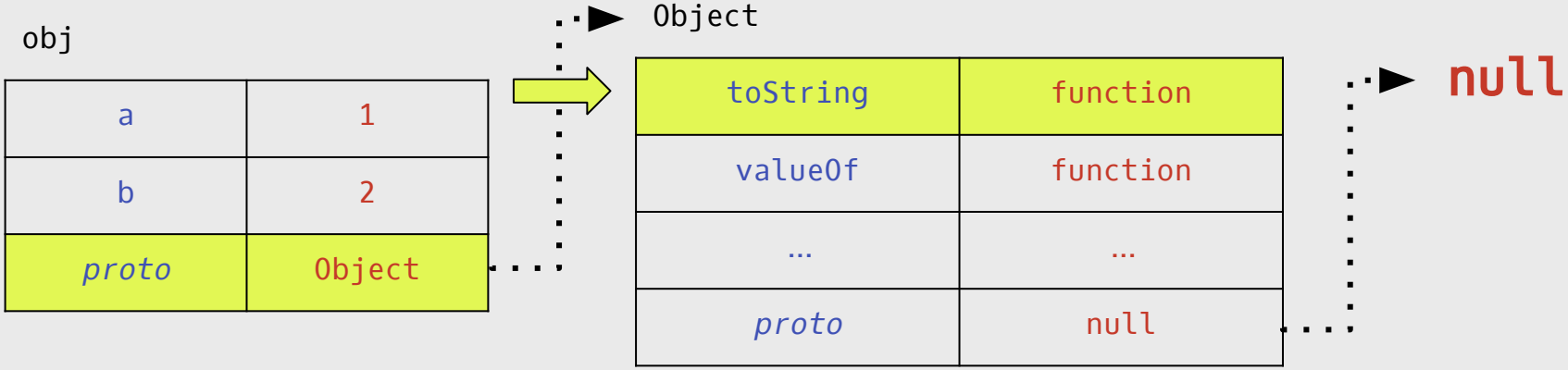
Object

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



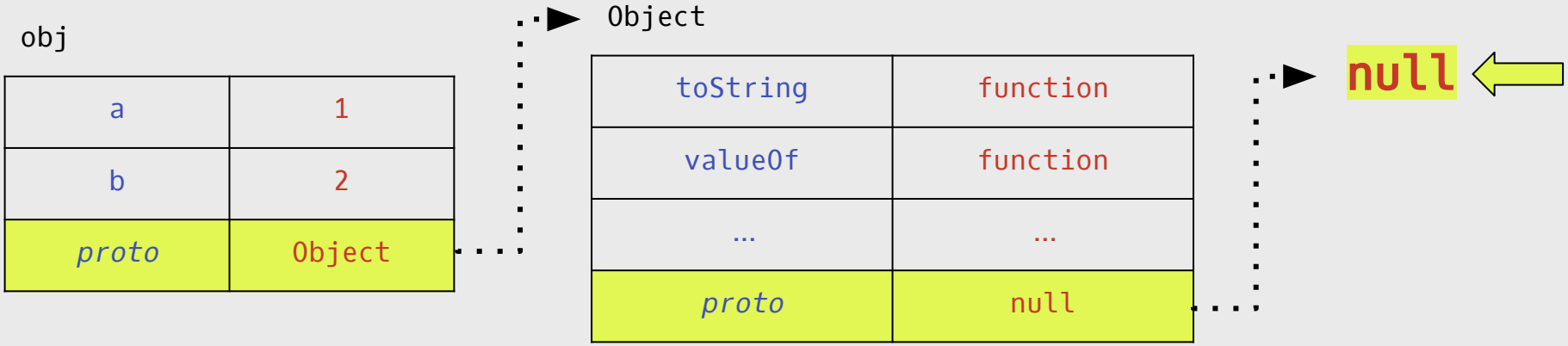
Object

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



Object

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



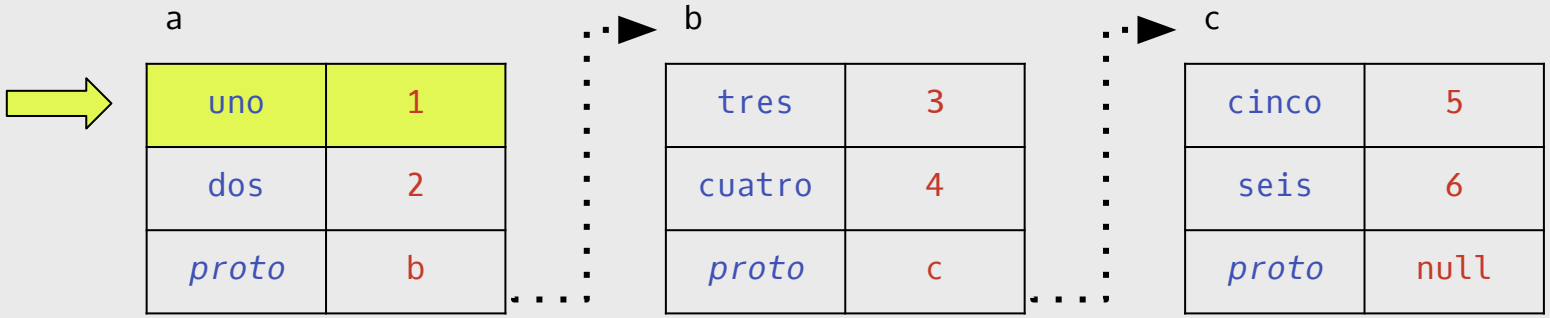
defineProperty

Si P es prototipo de A…

- Todas las propiedades de P son visibles en A
- Todas las propiedades del prototipo de P son visibles en A
- Todas las propiedades del prototipo del prototipo de P son visibles en A
- …

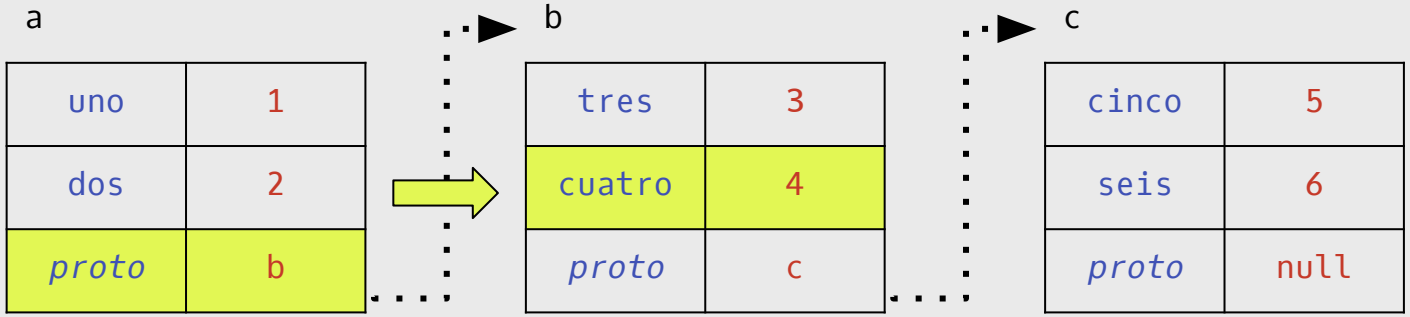
Object

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



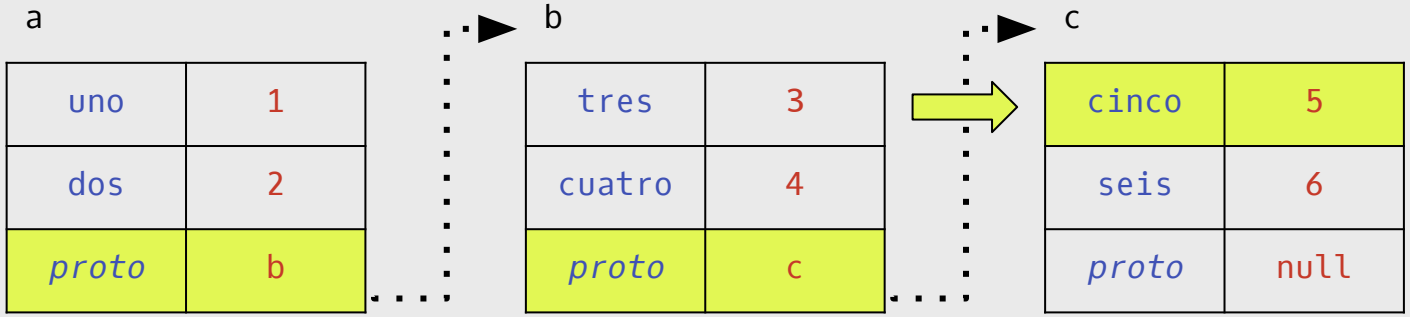
Object

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



Object

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



Object.create

Object

Object.create(*proto*, *properties*)

Genera un nuevo objeto

- *proto*: prototipo del objeto
- *properties*: descriptores de propiedades

Object.create()

```
let proto = {c: 3}

let obj = Object.create(proto)

obj.a = 1
obj.b = 2

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

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 II

¿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' }; // prototype

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

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

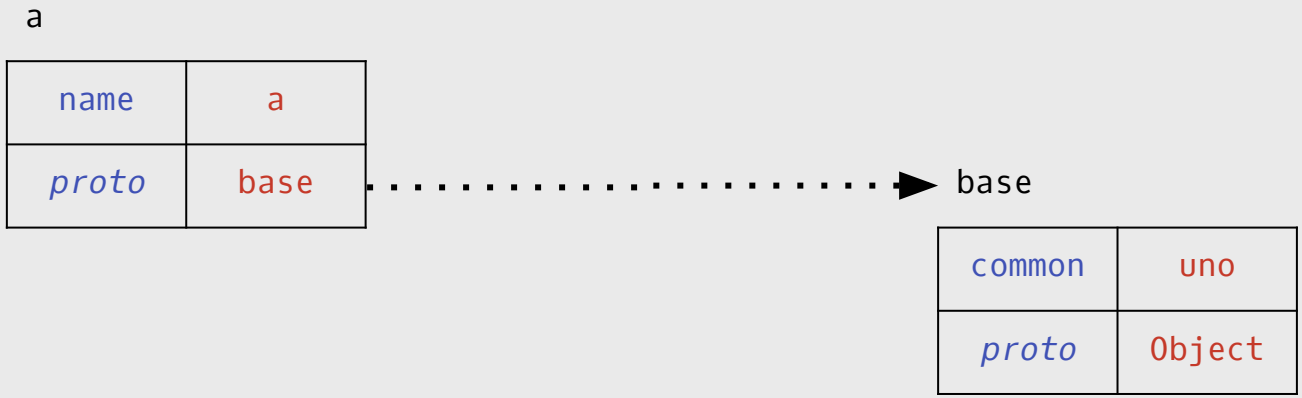
Object

```
base.common = 'dos';

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

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

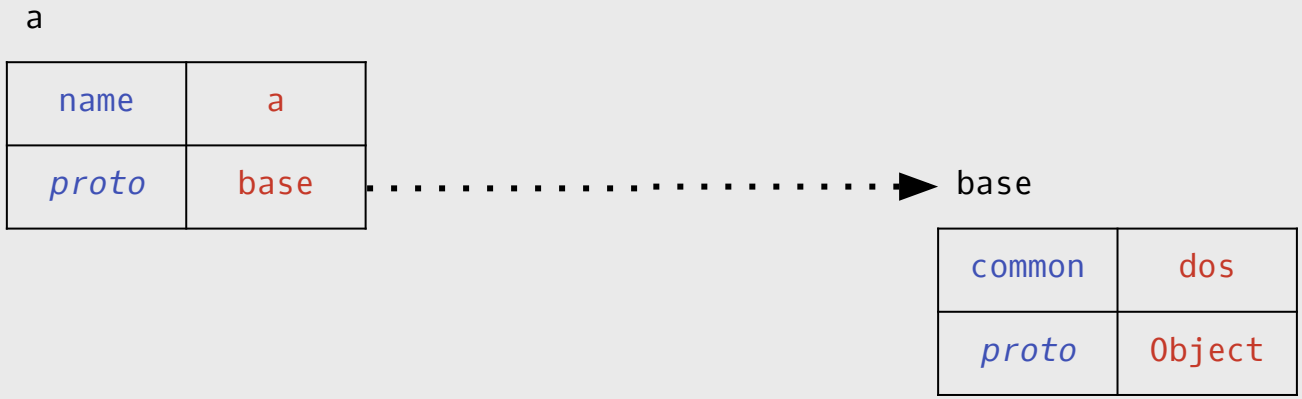
Object

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

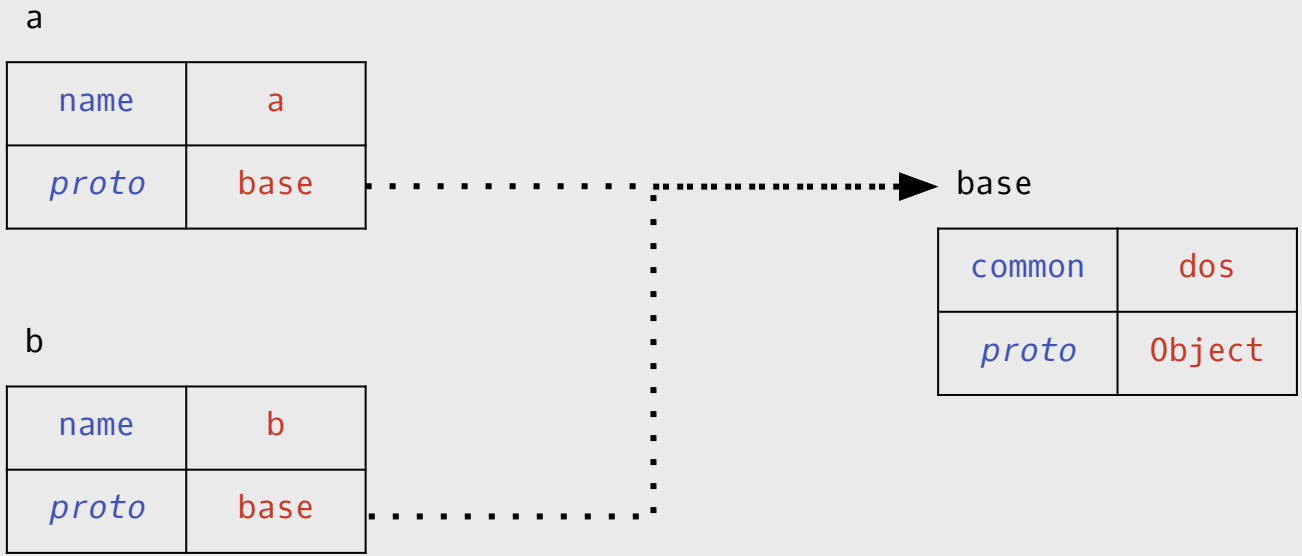
Object



Object



Object


a
name
a
proto
base
b
name
b
proto
base
base
common
dos
proto
Object

Object

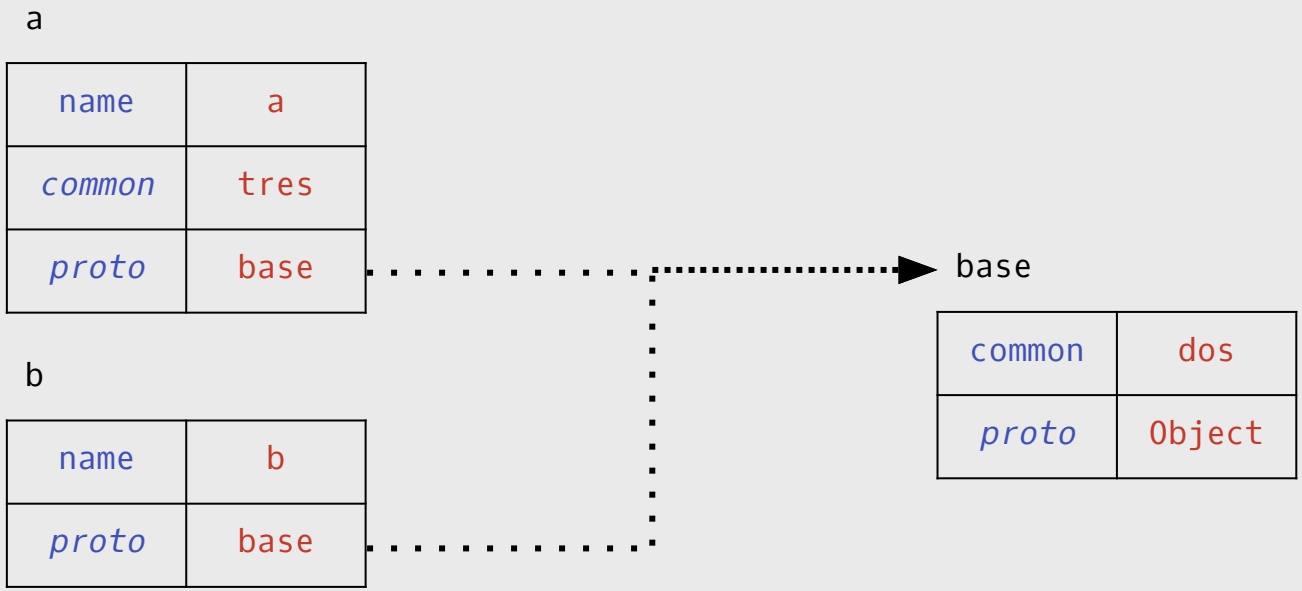
```
a.common = 'tres';
b.common; // ???
```

Object

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

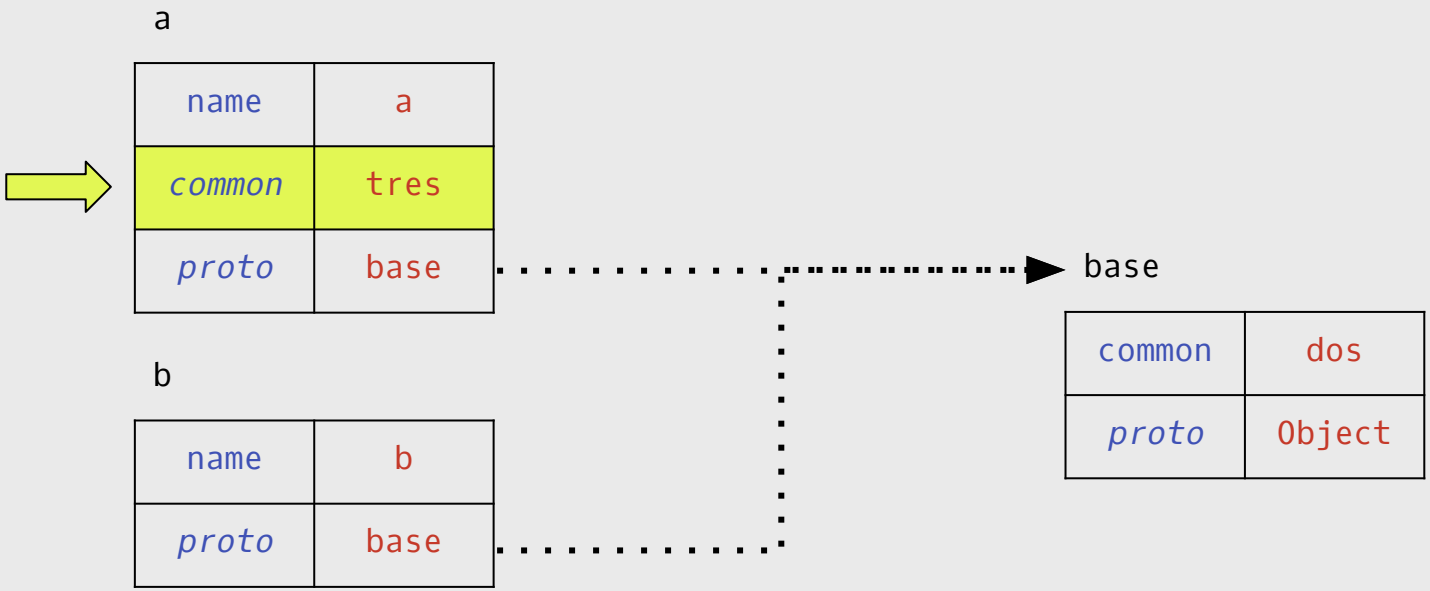
Object

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



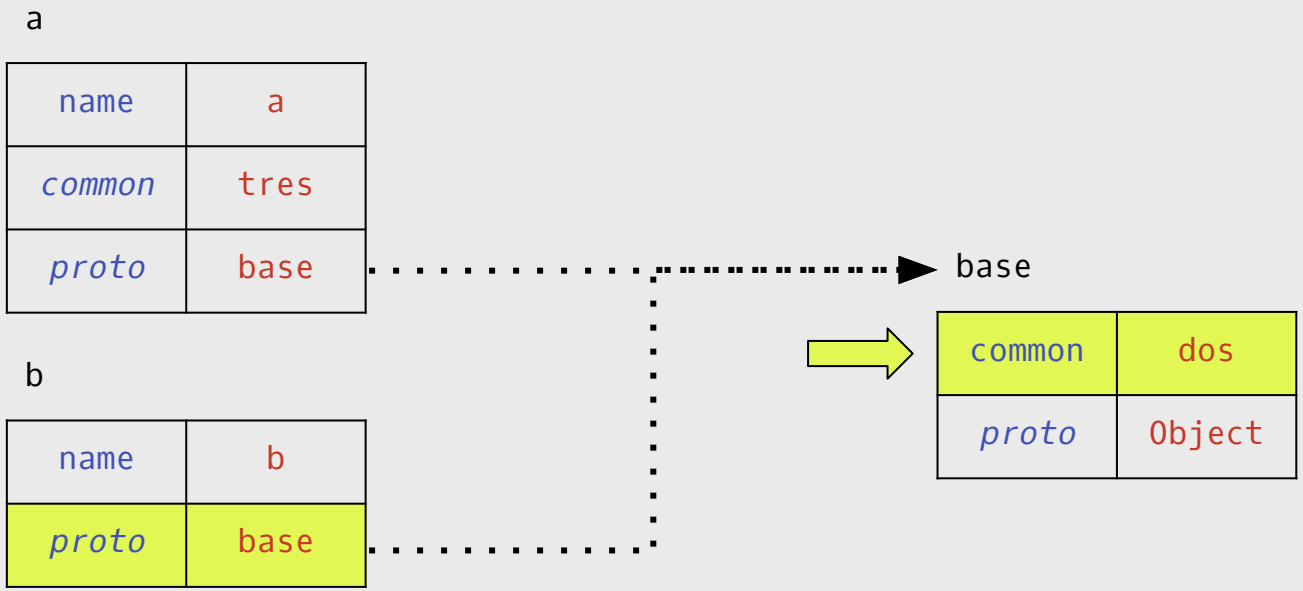
Object

b.common



Object

a.common



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.

Orientación a objetos

El contexto y los prototipos nos permite utilizar orientación a objetos en JavaScript.

- El contexto nos permite tener objetos únicos creados con un constructor
- Los prototipos nos permiten implementar mecanismo de herencia

Orientación a objetos

A día de hoy se utilizan clases para implementar código orientado a objetos.

- Son syntax sugar por encima de prototipos