

Fundamentos

Antes de empezar

Qué Necesitas

- Un editor de texto
- Ejecutar código javascript
 - Recomendado: node.js
 - Chrome

Hoisting

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()  
}
```

Scope

Qué es el scope

- El acceso que tenemos a variables y funciones en una zona específica de nuestro código

Qué es el scope

```
// Global Scope
function someFunction() {
  // Local Scope #1
  function someOtherFunction() {
    // Local Scope #2
  }
}
```

```
// Global Scope
function anotherFunction() {
  // Local Scope #3
}
// Global Scope
```

Ejemplo I

```
let name = 'Lorem'  
function logName () {  
    let name = 'Ipsum'  
    console.log(name)  
}  
logName()  
console.log(name)  
  
// ?  
// ?
```


Ejemplo I

```
let name = 'Lorem'  
function logName () {  
    let name = 'Ipsum'    <- El scope local tiene prioridad  
    console.log(name)  
}  
logName()  
console.log(name)  
  
// Lorem  
// Ipsum
```

Ejemplo II

```
let name = 'Lorem'  
function logName () {  
  name = 'Ipsum'  
  console.log(name)  
}  
logName()  
console.log(name)  
  
// ?  
// ?
```

Ejemplo II

```
let name = 'Lorem'  
function logName () {  
    name = 'Ipsum'  
    console.log(name)  
}  
logName()  
console.log(name)  
  
// Ipsum  
// Ipsum
```

Scope bloques if, for, etc.

```
if (true) {  
  let name = 'Lorem' // name está en el global scope  
}
```

```
console.log(name) // Error!
```

- Los bloques if, for y switch también crean scopes locales si usamos **let**

Scope bloques if, for, etc.

```
if (true) {  
  var name = 'Lorem' // name está en el global scope  
}
```

```
console.log(name) // Lorem
```

- Accesible por **hoisting**

Por qué limitar el scope?

- Es una buena práctica **tener acceso solo a lo que se necesita**: evitamos confusiones y errores innecesarios.
- **Evitamos choques** entre nombres de variables que usamos muchas veces en nuestro código: i, index, name, result, etc.

Scope: variables vs. funciones

- Las variables se tienen que crear antes de ser usadas y estar en el scope adecuado
- Las funciones son accesibles se creen antes o después, siempre que estén en el scope adecuado

```
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  
}
```

Scope: variables vs. funciones

- Const se comporta exactamente igual que let
- Salvo una excepción...

```
const uno = 1
```

```
const uno = 1;
```

```
uno = 2; // ERROR! Assignment to constant variable
```

Conclusión

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

Template strings

Permiten insertar variables dentro de una string fácilmente

- Se declaran usando **backticks** (``)
- Podemos meter **expresiones** dentro de \${} (variables, valores, etc.)

```
let error = 404  
let message = "Page not found"
```

```
let str= `Error ${error}: ${message}`  
console.log(str)           // Error 404: Page not found
```


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?*

Destructuring

Destructuring

Permite descomponer un array en varias variables:

```
[a, b] = [10, 20]  
console.log(a) // 10  
console.log(b) // 20
```

Destructuring

Permite descomponer un array en varias variables:

```
[a, b, ...rest] = [10, 20, 30, 40, 50]  
console.log(a) // 10  
console.log(b) // 20  
console.log(rest) // [30, 40, 50]
```

Rest operator: ...

- Permite capturar el resto de variables en forma de array

Destructuring

Podemos descomponer cualquier número de variables

```
[a] = [10, 20, 30, 40, 50]           // only first  
[a, b] = [10, 20, 30, 40, 50]       // only first and second
```

En cualquier posición

```
[, , a] = [10, 20, 30, 40, 50]       // only third  
[, , a, b] = [10, 20, 30, 40, 50]   // only third and fourth
```


Ejercicio

Cuanto vale tail en cada caso

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

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

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

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

Destructuring

Permite descomponer un objeto en varias variables:

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

Ejercicio

- Desestructura 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"
```

Destructuring

Podemos cambiar el nombre de las variables al desestructurarlas

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

Podemos desestructurar de forma anidada

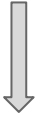
```
const { x: { y } } = { x: { y: 10 } }
```

```
// ?
```

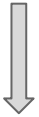
Destructuring

Podemos desestructurar de forma anidada

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



```
{ y } = { y: 10 }
```



```
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 } }
```


Ejercicio

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

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

Destructuring

El **rest operator** también permite recibir los argumentos de una función en forma de array

```
function consoleLogEverything(...things){  
  for(let thing of things){  
    console.log(thing)  
  }  
}
```

```
consoleLogEverything(1, 'dos', true)  
// 1  
// dos  
// true
```

Destructuring

El operador opuesto a **rest** es **spread**

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

```
let nums = [1, 2]  
suma(...nums)
```

Representa todos los elementos de un array, pero **de uno en uno**

Se puede utilizar para convertir un array en parámetros

Ejercicio destructuring

- Crea una función que reciba un número **cualquiera** de argumentos numéricos y devuelva el más pequeño y el más grande
- Llama a la función y captura los dos valores en variables separadas utilizando destructuring

Destructuring

Podemos emplear destructuring en los parámetros de una función

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

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

Destructuring

Podemos emplear destructuring en los parámetros de una función

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

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

Clausuras

Clausuras

```
function CrearNombreySaludo() {  
  name = 'Maria'  
  function saludar() {  
    console.log('Hola ' + name)  
  }  
}
```

```
saludar() // ?
```


Clausuras

```
function CrearNombreYSaludo () {  
  name = 'Maria'  
  return function () {  
    console.log('Hola ' + name)  
  }  
}
```

```
let saludar = CrearNombreYSaludo() // Que hay aqui?  
saludar()
```

Clausuras

```
function CrearNombreYSaludo () {  
  name = 'Maria'  
  return function () {  
    console.log('Hola ' + name)  
  }  
}
```

```
let saludar = CrearNombreYSaludo()  
saludar() // Que imprime esto?
```

Clausuras

```
function CrearNombreYSaludo () {  
  name = 'Maria'  
  return function () {  
    console.log('Hola ' + name)  
  }  
}
```

```
let saludar = CrearNombreYSaludo()  
saludar() // Hola Maria
```

Clausuras

```
function CrearNombreYSaludo () {  
  name = 'Maria'  
  return function () {  
    console.log('Hola ' + name)  
  }  
}
```

```
let saludar = CrearNombreYSaludo()  
saludar()
```

Ejercicio clausuras

- Crea una función que reciba un número y devuelva una función
- Esa función recibe un único número por parámetro y devuelve el resultado de sumar ambos números

Clausuras

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

Clausuras

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

Clausuras

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

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


Clausuras

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

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

Clausuras

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

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

Clausuras

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

```
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();
```

c1

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

```
i = 0
```

Clausuras

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

c1

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

```
i = 1
```

Clausuras

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

c1

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

~~i = 1~~

Clausuras

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

c1

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

```
i = 0
```

Clausuras

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

c1

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

```
i = 0
```

Clausuras

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

Variable libre

```
function counter() {  
  let i = 0;  
  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();  
let i = 10;  
console.log(c1()); // ???  
console.log(i); // ???
```

c1

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


Clausuras

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

c1

() => i++;

i = ??

Clausuras

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

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

Clausuras

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

Ejercicio

- Vuelve a resolver este bug, pero **sin utilizar let**

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



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]
```

Object

Object

- Un conjunto dinámico de propiedades
 - nombre: `string` o [symbol](#)
 - valor: cualquier valor
- Puede heredar propiedades de otro objeto
- Manejado por **referencia**

Creación de objetos

```
const obj = {};
```

```
const obj2 = { prop: 1 };
```

Object

Cuántas propiedades?

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


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.

Demo

Ejercicio

- Escribe una función **reclone**, una versión recursiva de clone

```
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 }
```

```
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' }
})
```


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.defineProperty(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.defineProperty(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`

Get

```
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

```
const obj = {};
```

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

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

Set

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

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

Prototipos

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

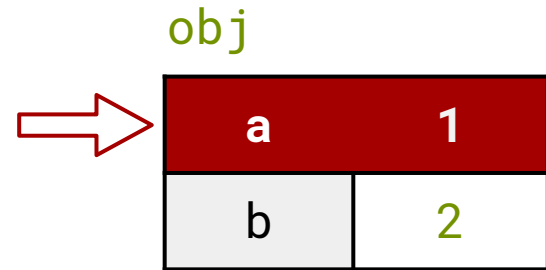
```
const obj = { a: 1, b: 2 };
```

obj

| | |
|---|---|
| a | 1 |
| b | 2 |

Object

`obj.a // 1`



Object

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

`obj`

| | |
|---|---|
| a | 1 |
| b | 2 |



???

Object

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

obj

| | |
|-------|--------|
| a | 1 |
| b | 2 |
| proto | Object |

Object

| | |
|----------|----------|
| toString | function |
| valueOf | function |
| ... | ... |
| proto | null |

► null

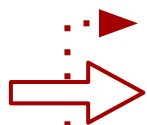
Object

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

obj

| | |
|---------------------|---|
| a | 1 |
| b | 2 |
| proto Object | |

Object



| | |
|-----------------|-----------------|
| toString | function |
| valueOf | function |
| ... | ... |
| proto | null |

► null

Object

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

obj

| | |
|---------------------|---|
| a | 1 |
| b | 2 |
| <i>proto</i> Object | |

Object

| | |
|-------------------|----------|
| toString | function |
| valueOf | function |
| ... | ... |
| <i>proto</i> null | |

null

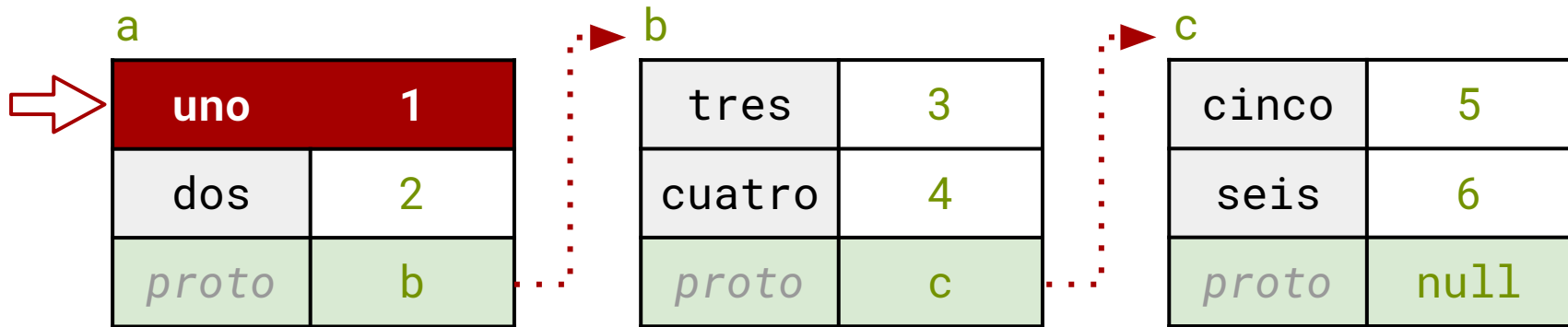


Object

- 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; // ???

Object

a

| | |
|--------------|------|
| name | a |
| <i>proto</i> | base |



base

| | |
|--------------|--------|
| common | uno |
| <i>proto</i> | Object |

Object

a

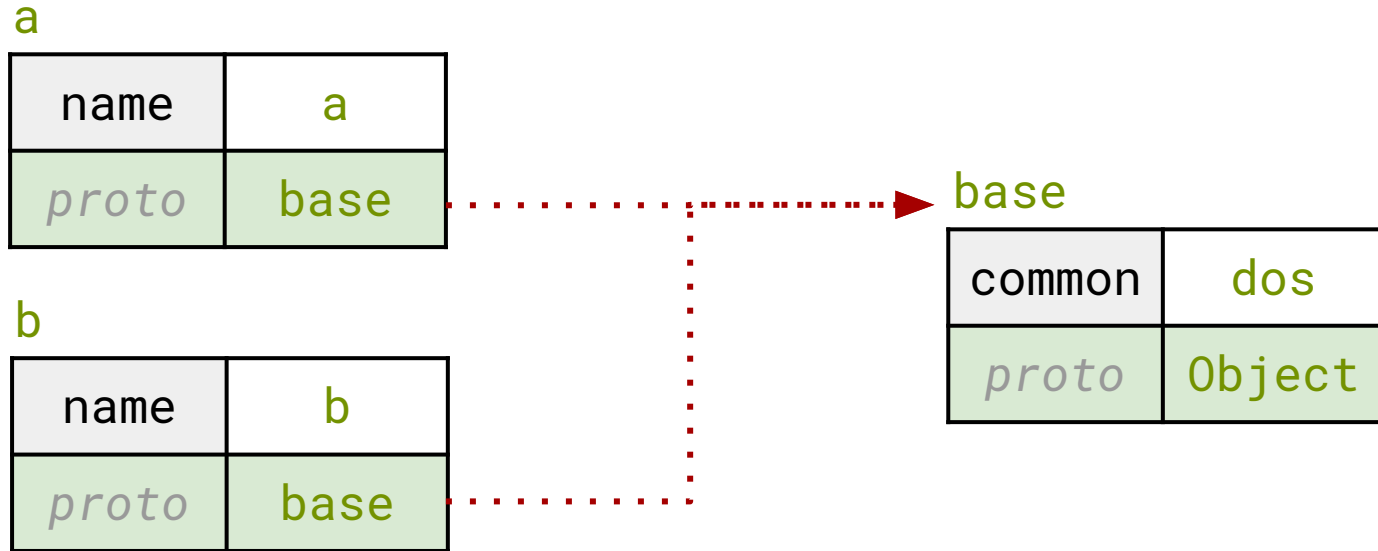
| | |
|--------------|------|
| name | a |
| <i>proto</i> | base |



base

| | |
|--------------|--------|
| common | dos |
| <i>proto</i> | Object |

Object



Object

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

Object

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

Object

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

a

| | |
|--------|------|
| name | a |
| common | tres |
| proto | base |

b

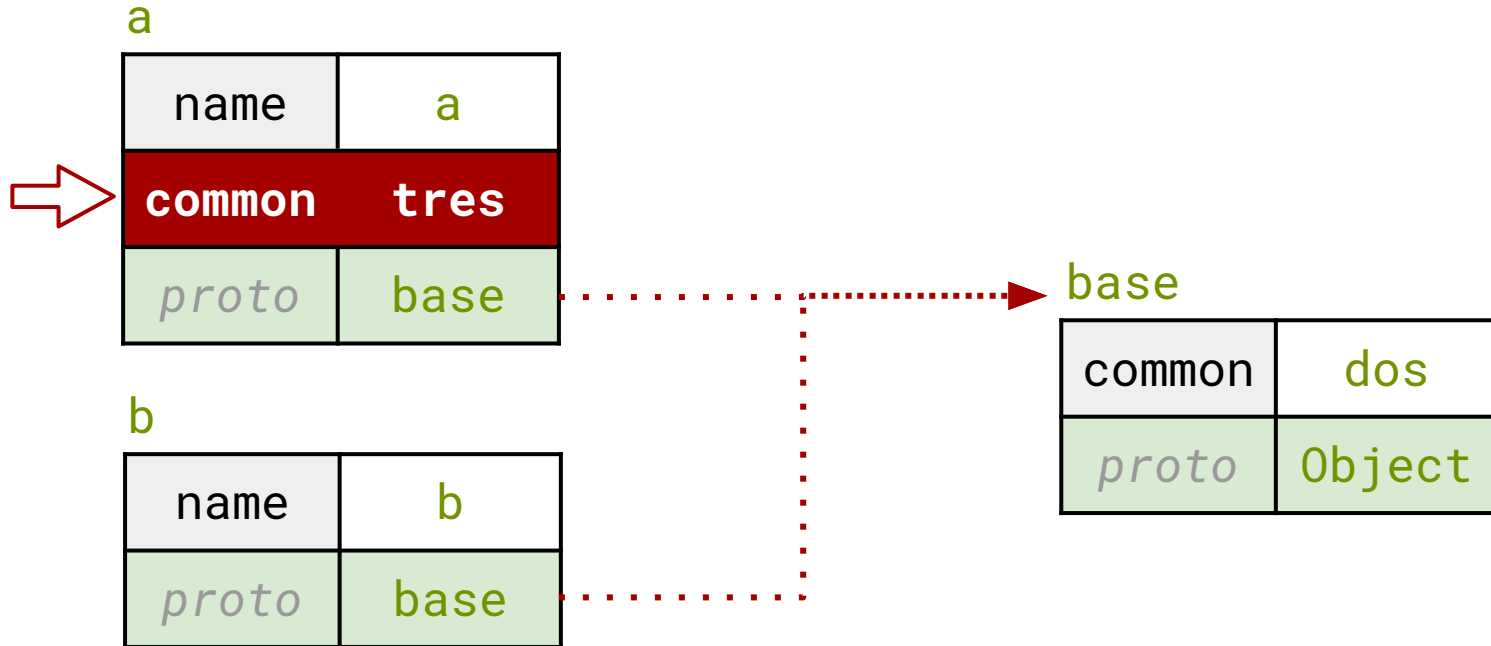
| | |
|-------|------|
| name | b |
| proto | base |

base

| | |
|--------|--------|
| common | dos |
| proto | Object |

Object

a.common



Object

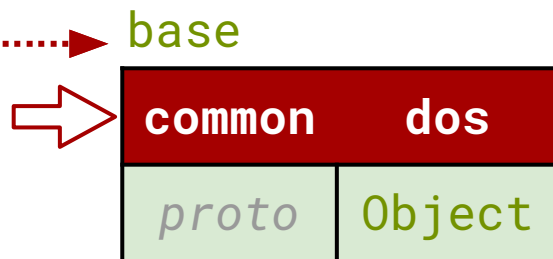
b.common

a

| | |
|--------------|------|
| name | a |
| common | tres |
| <i>proto</i> | base |

b

| | |
|--------------|-------------|
| name | b |
| proto | base |



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);
```

todo

| | |
|--------------|------|
| <i>proto</i> | base |
|--------------|------|



lista

| | |
|--------------|--------|
| items | [] |
| <i>proto</i> | Object |

Object

```
this.items.push(e1);
```

todo

| | |
|--------------|------|
| <i>proto</i> | base |
|--------------|------|



lista

| | |
|--------------|--------|
| items | [] |
| <i>proto</i> | Object |

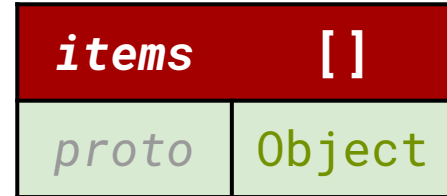
Object

```
this.items.push(e1);
```

`todo`



`lista`



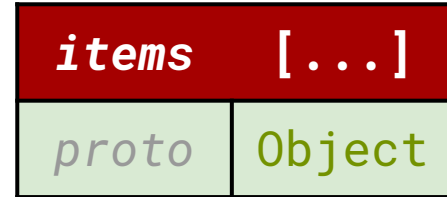
Object

```
this.items.push(e1);
```

todo



lista



Object

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

todo

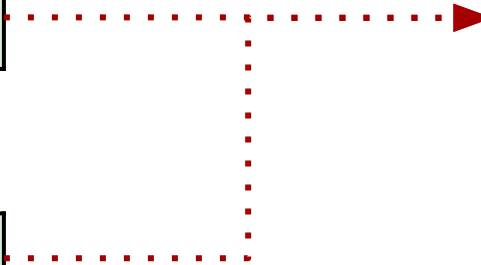
| | |
|--------------|------|
| <i>proto</i> | base |
|--------------|------|

compra

| | |
|--------------|------|
| <i>proto</i> | base |
|--------------|------|

lista

| | |
|--------------|--------|
| items | [...] |
| <i>proto</i> | Object |





Orientación a objetos

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()
```

```
// Imprime el objeto al que pertenece 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() // ?`

`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() // ?
```

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

- 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()  
  }  
}
```

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

Call, apply, bind

```
let andrea = {name: "Andrea"}
let obj = {
  name: "Alberto",
  presentarse: function(apellido1, apellido2){
    console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
  }
}

obj.presentarse("Arrabal", "Espada") //?
```

Call

```
let andrea = {name: "Andrea"}
let obj = {
  name: "Alberto",
  presentarse: function(apellido1, apellido2){
    console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
  }
}

obj.presentarse.call(andrea, "Arrabal", "Espada") // Hola, soy Andrea [...]
```


Apply

```
let andrea = {name: "Andrea"}
let obj = {
  name: "Alberto",
  presentarse: function(apellido1, apellido2){
    console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
  }
}

obj.presentarse.apply(andrea, ["Arrabal", "Espada"])
```

Ejercicio

Arregla este código **sin modificar func ni obj**

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

```
let obj = {  
    callFun : func  
}
```

```
obj.callFun.func()
```

Ejercicio

- Crea una función que reciba los parámetros **context** y **func**
- Debe devolver una versión de func que se ejecute en el contexto adecuado
- Utiliza call o apply

Bind

```
let andrea = {name: "Andrea"}
let obj = {
  name: "Alberto",
  presentarse: function(apellido1, apellido2){
    console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
  }
}

let bound = obj.presentarse.bind(andrea)
console.log(bound("Arrabal", "Espada"))
```

Orientación a objetos

Orientación a objetos

- La OOP entiende la programación como una serie de **entidades** (objetos) que interactúan entre si
- Está muy extendida porque replica nuestra forma de ver el mundo
- Es especialmente útil para programar sistemas en los que verdaderamente hay varios actores que interactúan entre ellos

Encapsulación

- La premisa principal de la OOP es dividir el código en piezas pequeñas (objetos) **que gestionen su propio estado**

Encapsulación

- La premisa principal de la OOP es dividir el código en piezas pequeñas (objetos) **que gestionen su propio estado**

```
let rabbit = {  
  speed: 0,  
  run: function() {  
    this.speed = 10  
    console.log(`Estoy corriendo a ${this.speed} km/h`)  
  }  
}
```


Interfaces

- Estos objetos se comunican al exterior a través de una interfaz
- Una interfaz es un conjunto de métodos (funciones) que dan funcionalidad al objeto de forma abstracta
- Estos métodos “abstractos” ocultan la implementación

Interfaces

```
let rabbit = {  
  speed: 0,  
  run: function() {  
    this.speed = 10  
    console.log(`Estoy corriendo a ${this.speed} km/h`)  
  },  
  sit: function() {  
    this.speed = 0  
    console.log('Estoy sentado')  
  }  
}
```

```
rabbit.run()  
rabbit.sit()
```

¿Y si quiero modificar el estado desde fuera?

Interfaces

```
let rabbit = {  
  speed: 0,  
  run: function() {  
    this.speed = 10  
    console.log(`Estoy corriendo a ${this.speed} km/h`)  
  },  
  sit: function() {  
    this.speed = 0  
    console.log('Estoy sentado')  
  }  
}
```

Interfaces

```
let rabbit = {  
  speed: 0,  
  run: function() {  
    this.speed = 10  
    console.log(`Estoy corriendo a ${this.speed} km/h`)  
  },  
  sit: function() {  
    this.speed = 0  
    console.log('Estoy sentado')  
  },  
  setSpeed: function(speed) {  
    this.speed = speed  
  }  
}
```

Interfaces

```
let rabbit = {  
  speed: 0,  
  run: function() {  
    this.speed = 10  
    console.log(`Estoy corriendo a ${this.speed} km/h`)  
  },  
  sit: function() {  
    this.speed = 0  
    console.log('Estoy sentado')  
  },  
  setSpeed: function(speed) {  
    if (speed > 0) {  
      this.speed = speed  
    } else {  
      this.speed = 0  
    }  
  }  
}
```

**Y si quisiéramos crear 1000
conejos?**

???

```
let rabbit = {  
  speed: 0,  
  run: function() {  
    this.speed = 10  
    console.log(`Estoy corriendo a ${this.speed} km/h`)  
  },  
  sit: function() {  
    this.speed = 0  
    console.log('Estoy sentado')  
  },  
  setSpeed: function(speed) {  
    if(speed > 0) {  
      this.speed = speed  
    } else {  
      this.speed = 0  
    }  
  }  
}
```


Construcción (cutre)

```
function makeRabbit(){
  return {
    speed: 0,
    run: function(){
      this.speed = 10
      console.log(`Estoy corriendo a ${this.speed} km/h`)
    },
    sit: function(){
      this.speed = 0
      console.log('Estoy sentado')
    },
    setSpeed: function(speed){
      if(speed > 0){
        this.speed = speed
      } else {
        this.speed = 0
      }
    }
  }
}
```

```
Let rabbit = makeRabbit()
```

Construcción (cutre)

```
function makeRabbit(name) {  
  
  // Constructor  
  let obj = { speed: 0, name: '' }  
  obj.name = name  
  
  // Metodos  
  obj.run = function() { ... }  
  obj.sit = function() { ... }  
  obj.setSpeed = function() { ... }  
  
  return obj  
}
```

- ¿Con qué mecanismo podríamos gestionar la herencia?

Construcción Javascript

```
function Rabbit(name) {  
    this.name = name  
    this.speed = 0  
}
```

```
Rabbit.prototype.run = function() { ... }  
Rabbit.prototype.sit = function() { ... }  
Rabbit.prototype.setSpeed = function() { ... }
```

```
let mordisquitos = new Rabbit("mordisquitos")  
let tambor = new Rabbit("tambor")
```

Constructores

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.`);  
}
```

```
toby.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  
}
```

Ejercicio

- Escribe la función **myNew** que replique el comportamiento de **new** utilizando **Object.create**
- **const toby = myNew(Dog, "Toby", arg2, arg3....)**

Ejercicio extra

- 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 Dog) // ???  
console.log(toby instanceof Animal) // ???  
console.log(toby instanceof Object) // ???  
  
console.log(Dog instanceof Animal) // ???  
console.log(Dog instanceof Function) // ???
```

Ejercicio

- Partiendo de la clase **Container** (next slide)
 - Vamos a implementar una serie de constructores derivados

```
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
```

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 los ejercicios anteriores
 - **ItemBox, ItemContainer, NestedContainer, Item y Container**

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
```

```
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)
```

