

ES6

Requisitos

- editor de texto
- **npm** y **node.js** instalados
- **git** para clonar el repo
- **navegador web**

Introducción

- ¿Dónde puedo **ejecutar ES6**?
 - En un navegador moderno
 - En una versión reciente de node.js
 - <https://kangax.github.io/compat-table/es6/>

Introducción

- **Transpilar ES6 a ES5**
 - **babel**
 - no es *necesario* si el entorno soporta ES6 nativo
 - pero es *muy útil*
 - ES7, ES8,...
 - JSX

Introducción

- Vamos a **preparar nuestro entorno de trabajo**
 - crea un **directorio**
 - abre la terminal y entra en ese directorio

Introducción

```
$ npm init
```

```
$ npm install -S babel-cli
```

Introducción

- Crea el fichero **test.js** y escribe, por ejemplo:

```
const nombre = 'Elias'  
console.log(`Hola, ${nombre}!`)
```

Introducción

```
$ ./node_modules/.bin/babel test.js
```


Introducción

```
$ npm install -S babel-preset-es2015
```

```
$ ./node_modules/.bin/babel --presets babel-preset-es2015 test.js
```

Introducción

```
$ ./node_modules/.bin/babel --presets babel-preset-es2015 test.js -o test.es5.js
```

```
{
  "name": "workspace",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/babel --presets babel-preset-es2015 test.js -o test.es5.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-es2015": "^6.24.1",
    "webpack": "^3.10.0"
  }
}
```

```
{
  "name": "workspace",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/babel --presets babel-preset-es2015 test.js -o test.es5.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-es2015": "^6.24.1",
    "webpack": "^3.10.0"
  }
}
```

Introducción

```
$ npm run build
```

String templates

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

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



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

Ejercicio

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

Ejercicio

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

```
var 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

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

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



```
var { x, y } = { x: 10, y: 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!*)

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

```
console.log(a, b) // "2 1"
```

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

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

Ejercicio

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

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

Ejercicio

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

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

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



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

Ejercicio

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

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

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

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

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

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

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

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

```
lista1 === lista2 // ??
```

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

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

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

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

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

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

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

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

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



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

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

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

```
suma() // 2
```

```
suma(2) // 3
```

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

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

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

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

sumaTodos(1)

sumaTodos(1, 1, 1, 1)

Import & export

Export & Import

- Muy sencillo:
 - **export** permite exportar *nombres* y hacerlos visibles para otros ficheros
 - **import** permite importar *nombres* desde otros ficheros

```
cuatro.js:  
export default 4
```

```
index.js:  
import cuatro from './cuatro.js'
```

numbers.js:

```
export const uno = 1
```

```
export const dos = 2
```

index.js:

```
import { uno } from './numbers.js'
```


numbers.js:

```
export const uno = 1
```

```
export const dos = 2
```

index.js:

```
import * as numbers from './numbers.js'
```

```
numbers.uno
```

```
numbers.dos
```

Ejercicio

- Crea un fichero **numbers.js** que exporte **uno**, **dos** y **tres**
- Crea un fichero **index.js** que los importe **TODOS** y loguee su valor por la consola

Import & Export

- Necesitamos una herramienta que...
 - resuelva los **import** y los **export**
 - y genere un **único fichero** con todas las dependencias que luego podemos utilizar en un navegador

Import & Export

```
$ npm install webpack
```

```
module.exports = {  
  entry: "./index.js",  
  output: {  
    filename: "bundle.js"  
  }  
}
```

Import & Export

```
$ ./node_modules/.bin/webpack
```

Ejercicio

- Crea un fichero **numbers.js** que exporte **uno**, **dos** y **tres**
- Crea un fichero **index.js** que los importe **TODOS** y loguee su valor por la consola

Import & Export

- **webpack**
 - resuelve los imports
 - **pero no compila ES6**
- **babel**
 - compila ES6
 - **pero no resuelve los imports**

Import & Export

```
$ npm install babel-loader
```

```
module.exports = {  
  entry: "./index.js",  
  output: {  
    filename: "bundle.js",  
  },  
  module: {  
    loaders: [{  
      test: /\.js$/,  
      exclude: /node_modules/,  
      loader: 'babel-loader',  
      query: {  
        presets: ['es2015']  
      }  
    }]  
  }  
}
```

```
import { uno } from './numbers.js'  
console.log(`Valor importado: ${uno}`)
```

```
console.log(`Valor importado: ${__WEBPACK_IMPORTED_MODULE_0__numbers_js__[ "a" ]}`)
```

Import & Export

```
$ ./node_modules/.bin/webpack --watch
```

```
{  
  // ...  
  
  "scripts": {  
    "build": "./node_modules/.bin/webpack",  
    "start": "./node_modules/.bin/webpack --watch"  
  }  
  
  // ...  
}
```

Import & Export

```
$ npm run start
```

Import & Export

- Para tener más ordenado el código, vamos a crear dos directorios
 - **/src**: aquí meteremos nuestro código fuente
 - **/dist**: aquí meteremos el **bundle.js**

Import & Export

- `/es6/001`

```

const path = require('path');

module.exports = {
  context: path.resolve(__dirname, 'src'),
  entry: 'index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  resolve: {
    modules: [
      path.resolve(__dirname, 'src'),
      'node_modules'
    ]
  },
  module: {
    loaders: [{
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'babel-loader',
      query: {
        presets: ['es2015']
      }
    }]
  }
}

```

```
const path = require('path');

module.exports = {
  context: path.resolve(__dirname, 'src'),
  entry: 'index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  resolve: {
    modules: [
      path.resolve(__dirname, 'src'),
      'node_modules'
    ]
  },
  module: {
    loaders: [{
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'babel-loader',
      query: {
        presets: ['es2015']
      }
    }]
  }
}
```

```
const path = require('path');

module.exports = {
  context: path.resolve(__dirname, 'src'),
  entry: 'index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  resolve: {
    modules: [
      path.resolve(__dirname, 'src'),
      'node_modules'
    ]
  },
  module: {
    loaders: [{
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'babel-loader',
      query: {
        presets: ['es2015']
      }
    }]
  }
}
```

```
import { uno } from './numbers.js'
```

```
import { uno } from 'numbers'
```

```
import { uno as one } from 'numbers'
```

Ejercicio

- Crea un fichero **numbers.js** que exporte **uno**, **dos** y **tres**
- Crea otro fichero **chars.js** que exporte **a**, **b**, y **c**
- Crea un fichero **numbers_and_letters.js** que exporte dos nombres:
 - **letters**, con **todas** las letras importadas de **chars.js**
 - **numbers**, con **todos** los números importados de **numbers.js**
- Modifica **index.js** para que importe y liste todos los números desde **numbers_and_letters.js**

Declaración de variables

Declaración de variables

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

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

myFunc()

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

myFunc()

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

myFunc()

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

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

Ejercicio

- Encuentra y arregla el bug

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


Ejercicio

- Encuentra y arregla el bug

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

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

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

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

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

```
const uno = 1
```

```
const uno = 1;
```

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

Clausuras


```
let a = 1
```

```
function what() {  
  return a  
}
```

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

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

```
let thing1 = what3()
```

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

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

Clausuras

- Las variables en JS tienen **alcance indefinido**
 - **NO** se destruyen cuando la ejecución abandona el ámbito de creación
 - **Solo se destruyen** cuando no se puede acceder a ellas (no hay ninguna referencia viva)

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

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

```
function what4() {  
  let i = 0  
  return i++  
}
```

```
console.log(what4())  
console.log(what4())  
console.log(what4())
```

```
function what5() {  
  let i = 0  
  return function() {  
    return i++  
  };  
}
```

```
let thing1 = what5()
```

```
thing1()
```

```
thing1()
```

```
thing1()
```

```
console.log(thing1())
```



```
thing1 = null
```

Ejercicio

- Encuentra y arregla el bug
 - (siguiente diapositiva)

```
let theThing = null;
let replaceThing = function() {
  let originalThing = theThing;
  theThing = {
    bigChunk: new Array(1e8).join('*'),
    someMethod: function() {
      if (originalThing) {
        console.log('originalThing exists');
      }
    }
  };
};

setInterval(replaceThing, 10);
```

Symbols

Symbols

- ES5 ofrece **6** tipos de datos primitivos
 - Boolean
 - Number
 - String
 - Symbol
 - Null
 - Undefined

Symbols

- ES5 ofrece **1** tipo de dato compuesto
 - **Object**

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

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

`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

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'
```

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

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

Iterators

```
let i = makeIterator([1, 2, 3, 4]);  
  
let next = i.next();  
while (!next.done) {  
  console.log(next.value);  
  next = i.next();  
}
```

Iterators

```
let i = makeIterator([1, 2, 3, 4]);  
  
for (let n = i.next(); !n.done; n = i.next()) {  
  console.log(n.value);  
}
```


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

- Decimos que un objeto es **iterable** cuando...
 - En su propiedad **Symbol.iterator**
 - Tiene una función
 - Que devuelve un iterador

Iterators

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

Iterators

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

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); // ???
```

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);  
}
```

Ejercicio

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

Ejercicio

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

Objetos

```
const k = 'a'
```

```
const obj1 = { [k]: 1 }
```

```
const obj2 = { [k]: 1 }
```

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



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

```
obj3.b = 2
```

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

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

```
obj3.b = 2
```

```
console.log(obj1.b)
```

Object.assign

Object

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

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

```
Object.assign(a, b)
```

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

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

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

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

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

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

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

```
const c = { c: 3 }
```

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

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


Ejercicio

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

Ejercicio

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

```
const u1 = { username: 'root', password: 'iamgod' }  
const u2 = { username: 'luser', password: '12345' }  
const users = { u1: u1, u2: u2 }  
  
const usersCopy = clone(users);  
usersCopy.u3 = { username: 'admin', password: 'aDS00Dkxx098Sd' }  
  
console.log(users.u3) // ???  
  
usersCopy.u1.username = 'p0wnd'  
  
console.log(users.u1.username) // ???  
  
users.u1 === usersCopy.u1 // ???
```

Ejercicio

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

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

Ejercicio

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

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

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

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

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



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

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

```
const x = [{ a: 1 }, [{ b: 2 }]]  
const y = [{ b: 2 }, [], { c: 'hi' }]  
  
console.log(merge(x, y))
```

Object.defineProperty

Object.defineProperty

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

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

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

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

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

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

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

```
const obj = {}  
Object.defineProperties(obj, {  
  b: { value: 2 },  
  c: { value: 3 }  
})  
  
console.log(Object.keys(obj)) // ????
```



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

```
const obj = {}
```

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

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

```
const obj = {}
```

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

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

Object

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

getters y setters

Object

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

Object

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

Object

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

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

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


Object

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

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

Object

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

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

Object

```
const obj = {};  
obj.fahrenheit = temp.fahrenheit;  
  
obj.celsius = -12.22;  
console.log(obj.fahrenheit); // ???
```

Ejercicio

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

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

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

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

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

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

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

Object

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

Ejercicio

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

Ejercicio

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

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

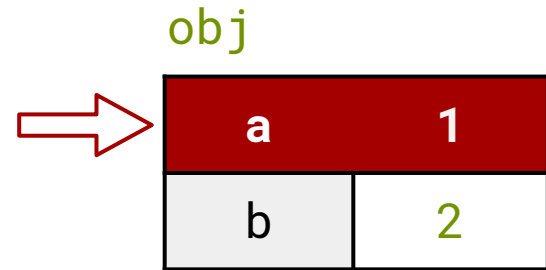
```
const obj = { a: 1, b: 2 };
```

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
proto Object	

Object

toString	function
valueOf	function
...	...
proto null	

null

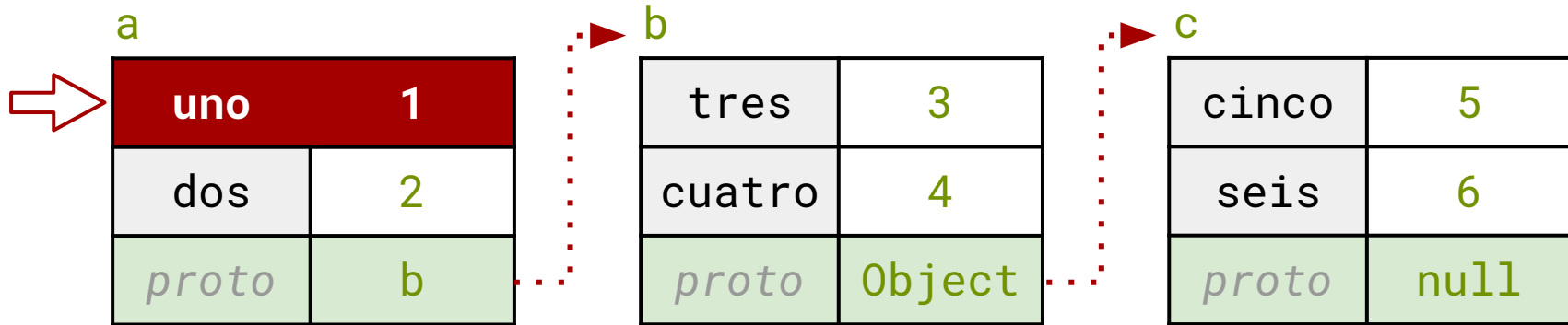


Object

- 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**
 -

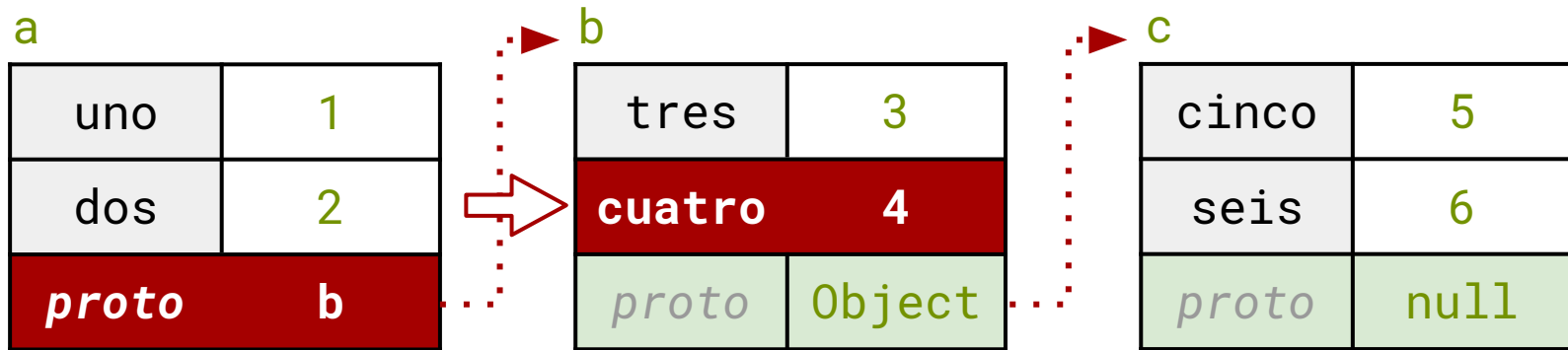
Object

a.uno // 1



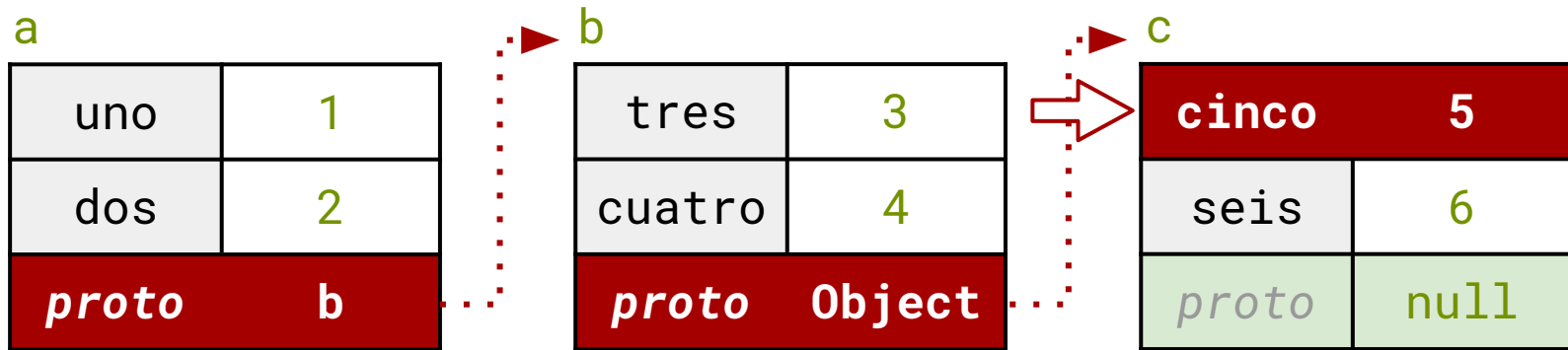
Object

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

Object

a.common; // ???

Object

a

name	a
<i>proto</i>	base



base

common	uno
<i>proto</i>	Object

Object

a

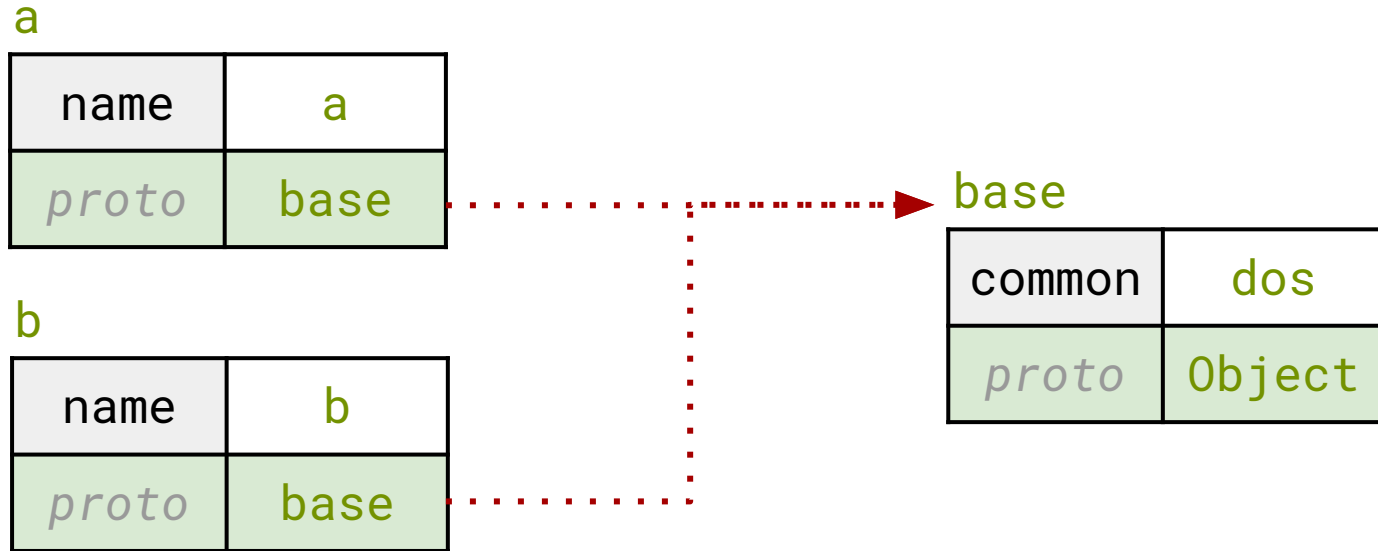
name	a
<i>proto</i>	base



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	<i>tres</i>
<i>proto</i>	base

b

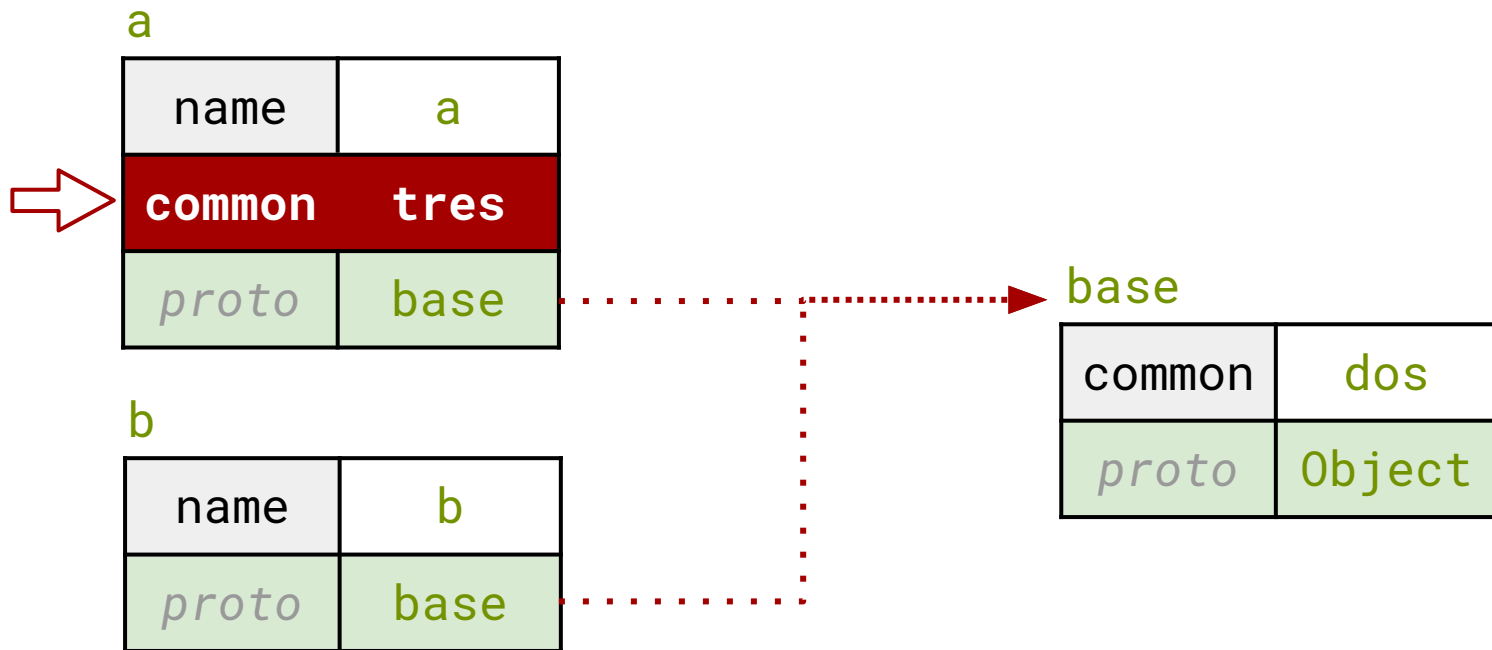
name	b
<i>proto</i>	base

base

common	dos
<i>proto</i>	Object

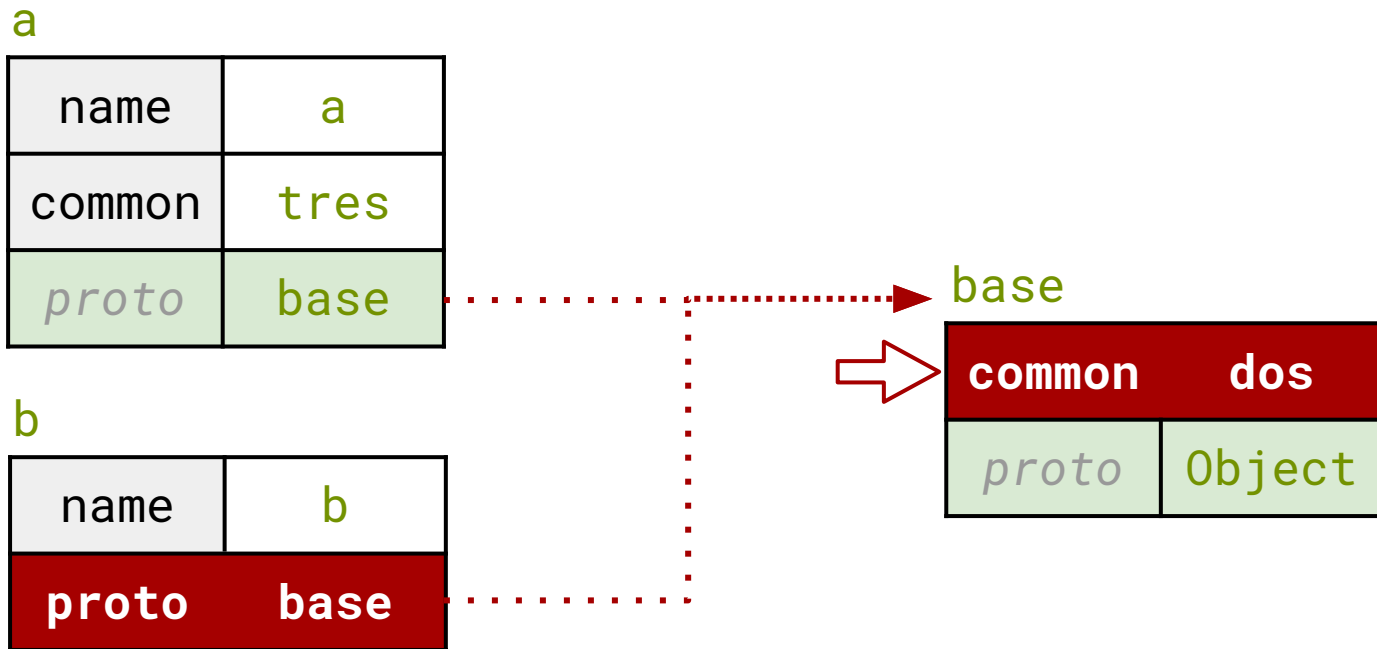
Object

a.common



Object

b.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

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

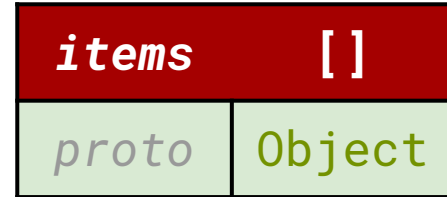
Object

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

`todo`



`lista`



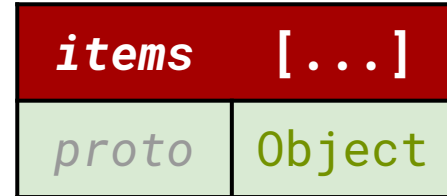
Object

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

todo



lista



Object

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

todo

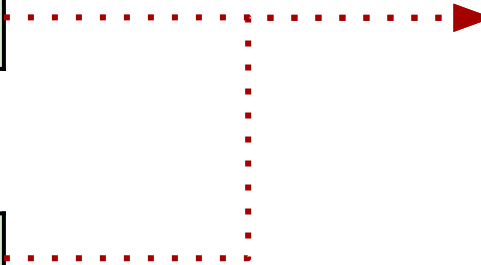
<i>proto</i>	base
--------------	------

compra

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

lista

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



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; // ???
```

receptor

```
const obj = { counter: 0 }
```

```
obj.increment = function() {  
  obj.counter++  
}
```

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

```
const obj = { counter: 0 }
```

```
obj.increment = function() {  
  obj.counter++  
}
```

```
const obj2 = { counter: 1 }
```

```
obj2.increment = obj.increment
```

```
console.log(obj2.counter) // ???
```

```
obj2.increment()
```

```
console.log(obj2.counter) // ???
```

Ejercicio

- Encuentra y arregla el bug en el ejemplo anterior

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que no tenga binding léxico
 - Que apunte al objeto “adecuado”

Programación Orientada a Objetos

¿Cuál es el objeto adecuado?

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que **no tenga binding léxico**
 - Que apunte al objeto que **está a la izquierda del punto**

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que **no tenga binding léxico**
 - Que apunte al objeto que **recibe el mensaje**

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que **no tenga binding léxico**
 - Que apunte al objeto que **recibe el mensaje**
 - Y se **vincule** en el momento de la **invocación**

Programación Orientada a Objetos

this

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

```
setInterval(obj.increment, 1000)
```

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

```
const inc = obj.increment
```

```
inc()
```

```
inc()
```

```
console.log(obj.counter)
```

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

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

```
user.greet()
```

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


Receptor

Teniendo:

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

¿Qué significa esto?

obj.nombre;

Receptor

Teniendo:

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

¿Y esto?

```
obj.saludo;
```

Receptor

Teniendo:

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

¿Y esto otro?

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

Receptor

Teniendo:

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

¿Es lo mismo?

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

NO

Receptor

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

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

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

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

Receptor

Cuatro maneras de invocar a una función:

1. Invocación directa

Receptor

Cuatro maneras de invocar a una función:

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


```
const counter = {  
  count: 0,  
  increment: function() { this.count++; }  
}  
  
$('#button').on('click', counter.increment)
```

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

```
obj.saludo()
```

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

Programación Orientada a Objetos

Cuatro maneras de invocar a una función:

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

Programación Orientada a Objetos

```
fn.call(context, arg1, arg2, ...)
```

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

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

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

saludo() // ???

obj1.saludo() // ???

saludo.call(obj1) // ???

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

```
const join = [].join
```

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

```
const letras = ['a', 'b', 'c']
```

```
join(numeros, ',') // ???
```

```
const concat = [].concat
```

```
const primero = [1, 2]
```

```
const segundo = [3, 4]
```

```
const tercero = [5, 6]
```


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

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

suma(1, 1) // ???

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

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

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

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

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

```
obj.saludo()
```

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

```
const obj = { nombre: 'Homer' }
```

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

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

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

```
const fn = saludo.call(obj)
```

```
fn.call(null) // ???
```

```
fn.call({ nombre: 'Fry' }) // ??
```

¿Qué hace esta función?

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

¿Qué hace esta función?

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

```
const algo = misterio();
```

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

¿Qué hace esta función?

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

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

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

¿Qué hace esta función?

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

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

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


¿Qué hace esta función?

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

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

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

¿Qué hace esta función?

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

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

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

¿Qué hace esta función?

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

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

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

¿Qué hace esta función?

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

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

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

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

¿Qué hace esta función?

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

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

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

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

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

```
obj.saludo()
```

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

```
obj.saludo()
```


arrow functions

Arrow functions

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

Arrow functions

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

Arrow functions

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

Arrow functions

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

Arrow functions

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

Arrow functions

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

Arrow functions

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

```
(arg1, arg2, ...) => expression;
```


Arrow functions

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

Arrow functions

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

Arrow functions

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

Arrow functions

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

```
(arg1, arg2, ...) => expression;
```

```
arg => expression;
```

Arrow functions

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

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

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

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

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

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

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

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



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

```
const obj = { obj: 'Homer' }
```

```
const binded = saludo.bind(obj)
```

```
binded() // ???
```

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

Constructores

Constructores

Cuatro maneras de invocar a una función:

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

Constructores

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

Constructores

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

Constructores

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

```
function Dog(name) {  
  this.name = name;  
}  
  
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}  
  
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}  
  
const toby = new Dog("Toby");  
toby.sit();
```



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

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

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

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

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

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

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

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

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

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

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

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

Constructores

¿Verdadero o falso?

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

Constructores

¿Verdadero o falso?

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

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

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

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

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

Constructores

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

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

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

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

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

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



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

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

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

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

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

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

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

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

Ejercicio

- Escribe un constructor **User** que reciba un nombre de propiedad y tenga el método **greet** que muestre un saludo con el nombre

Ejercicio

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

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

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

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

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

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

```
const homer2 = new User('Homer')  
console.log(homer === homer2) // ???  
  
console.log(homer.greet === homer2.greet) // ???  
  
homer2.greet = () => console.log('Buen dia')  
  
homer.greet() // ??? (POR QUE?)  
  
User.prototype.greet = () => console.log('Buen dia')  
fry.greet() // ???  
homer2.greet() // ???
```

Ejercicio

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

Ejercicio

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

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

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

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

```
CountedUser.getInstanceCount() // 2
```

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

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

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

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

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

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

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

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**

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

Ejercicio

- Reescribe **withCount** para clases

Promesas

```
const p = new Promise()
```

```
const p = new Promise(resolve => resolve(42))
```

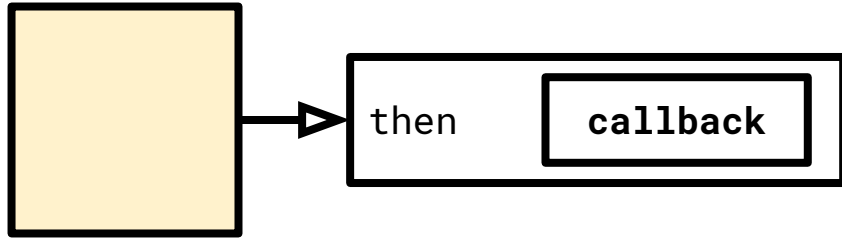


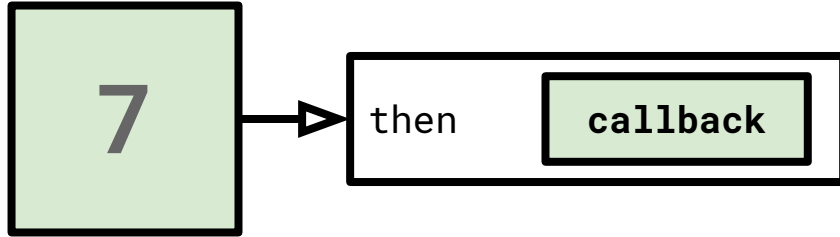
```
const p42 = new Promise(resolve => resolve(42))
```

```
console.log(p42) // ???
```

```
const p42 = new Promise(resolve => resolve(42))  
  
p42.then(value => console.log(value))
```

```
const p = new Promise(resolve => {})  
  
p.then(value => console.log('Nunca se ejecuta!'))
```





```
const p = new Promise((resolve) => {  
  setTimeout(() => resolve(42), 1000)  
})
```

```
p.then(value => console.log(`got ${value}, delayed!`))
```

```
const p = new Promise((resolve) => {  
  setTimeout(() => resolve(42), 1000)  
  setTimeout(() => resolve(84), 1500)  
})
```

```
p.then(value => console.log(`got ${value}, delayed!`))
```

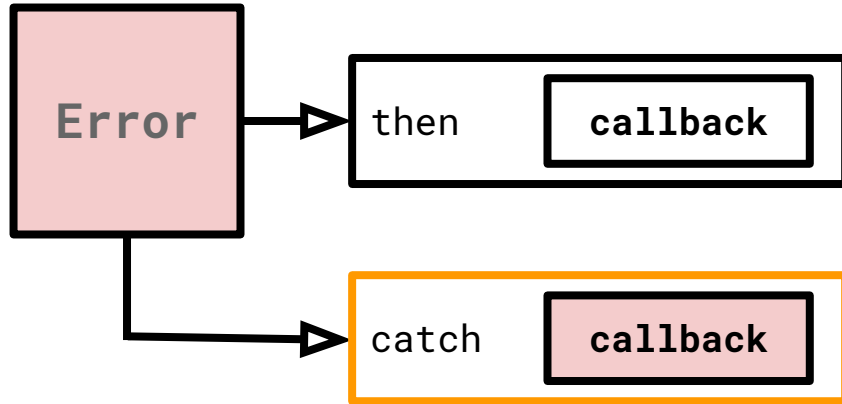
```
const p = new Promise((resolve, reject) => {  
  reject(new Error('Impossible'))  
}))
```

```
p.then(value => console.log(value))
```



```
const p = new Promise((resolve, reject) => {  
  reject(new Error('Impossible'))  
}))
```

```
p.catch(error => console.error(error))
```



Promesas

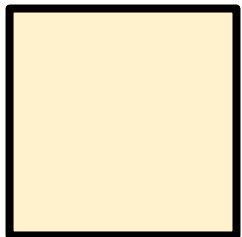
- Una promesa tiene tres estados:
 - pendiente
 - resuelta
 - rechazada
- Cuando una promesa se **resuelve** o se **rechaza**, **no puede volver a cambiar de estado**
 - se queda resuelta o rechazada para siempre

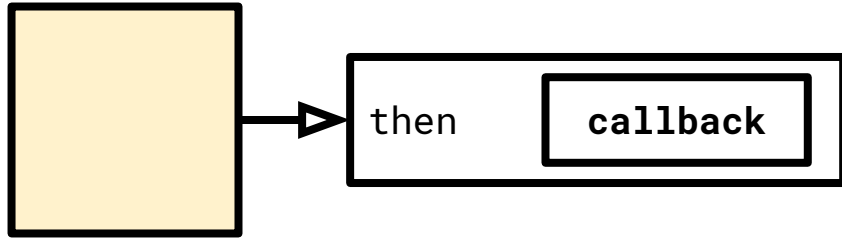
Ejercicio

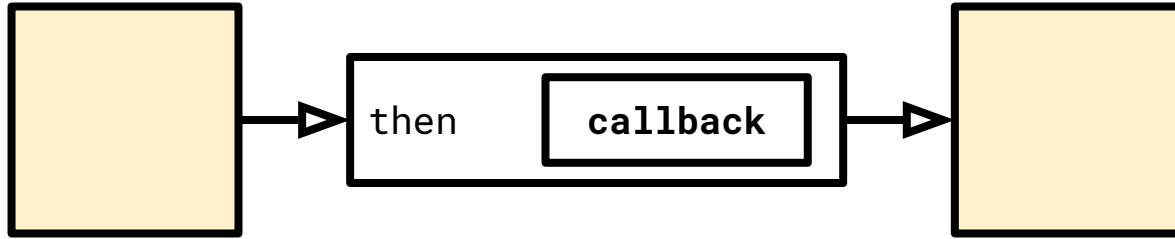
- Escribe una función **throwOneCoin** que devuelva una promesa que represente el lanzamiento de una moneda.
 - **50%** de las veces, la promesa se **resuelve** y se muestra **“cruz!”** por la consola
 - **50%** de las veces, la promesa se **rechaza** y se muestra **“cara...”** por la consola

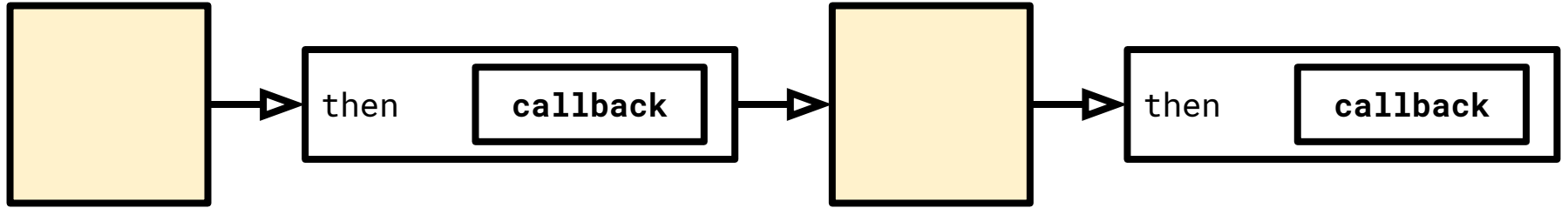
Promesas

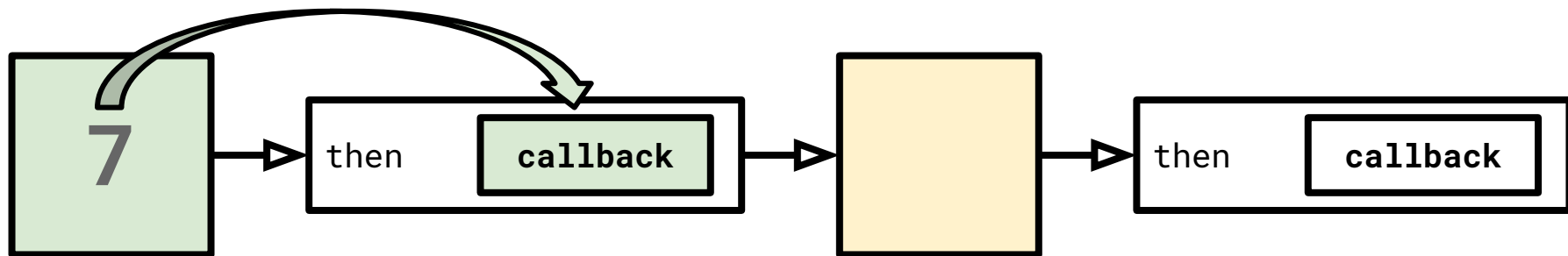
- Las llamadas a **.then()** y a **.catch()** devuelven **una nueva promesa**
- Que representa el **valor de retorno** de sus **callbacks**
- El callback the **.then()** se ejecuta cuando la promesa se **resuelve**
- El callback the **.catch()** se ejecuta cuando la promesa se **rechaza**

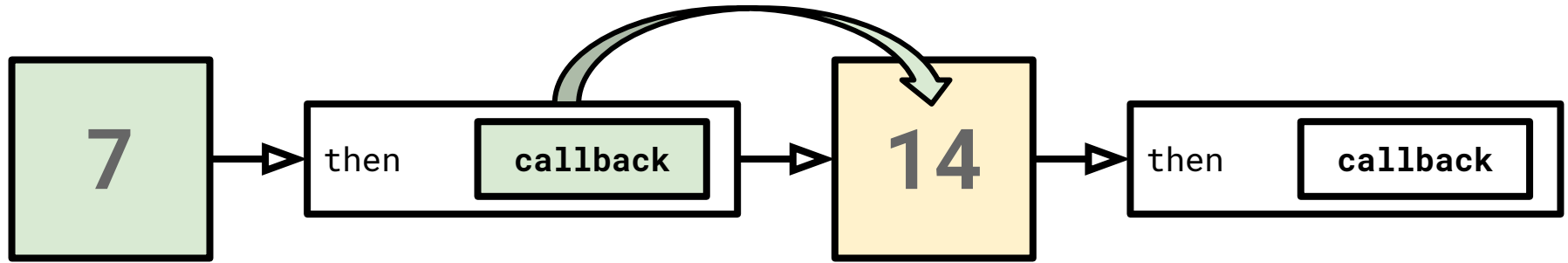


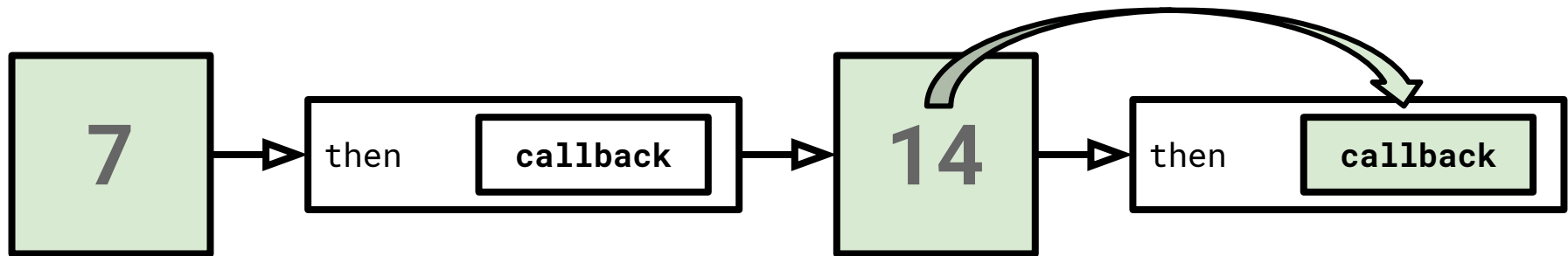












```
const promise = new Promise(resolve => resolve(42))
```

```
const promise2 = promise.then((value) => {  
    return value * 2  
})
```

```
const promise3 = promise2.then((value2) => {  
    console.log(value2) // ???  
})
```

```
const promise = new Promise((resolve) => {  
  setTimeout(() => resolve(42), 1000)  
})
```

```
const promise2 = promise.then((value) => {  
  return value * 2  
})
```

```
const promise3 = promise2.then((value2) => {  
  console.log(value2) // cuando se ejecuta?  
})
```

```
const promise = new Promise((resolve, reject) => {  
  setTimeout(() => reject(42), 1000) // WATCH OUT!  
}))
```

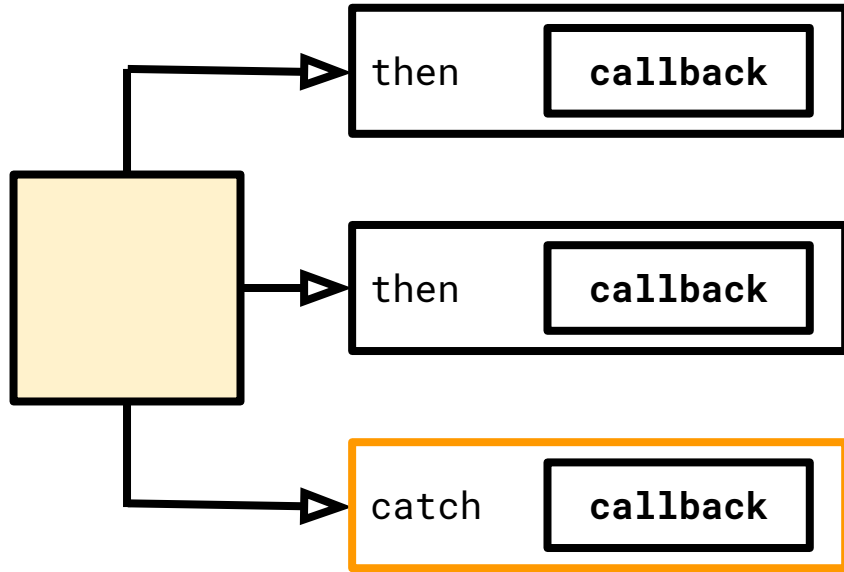
```
const promise2 = promise.then((value) => {  
  return value * 2  
}))
```

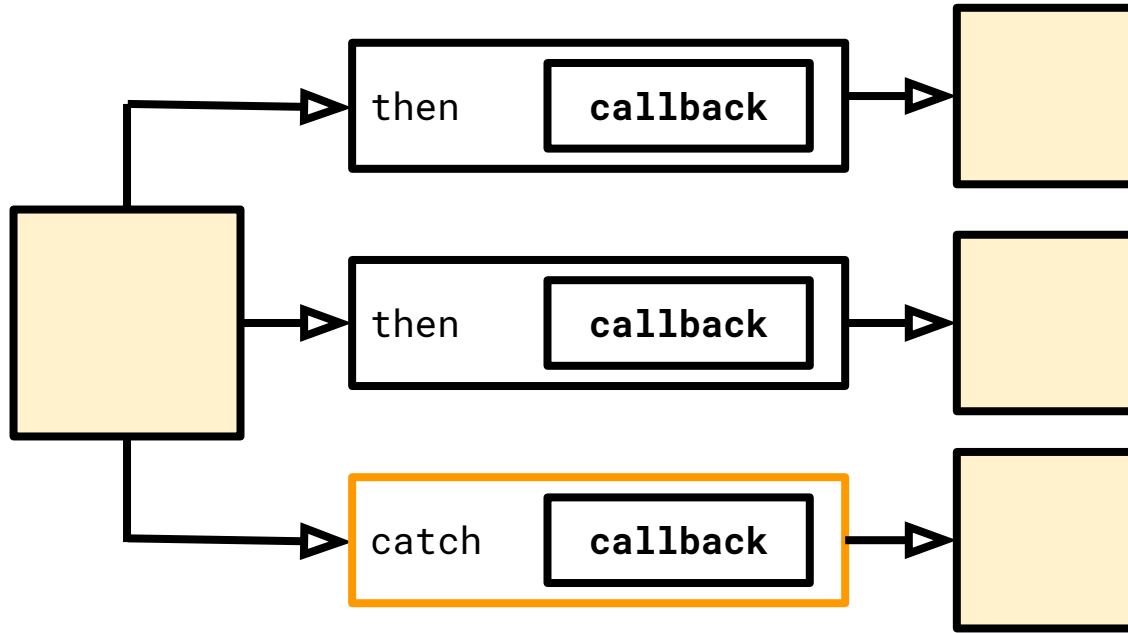
```
const promise3 = promise2.then((value2) => {  
  console.log(value2) // cuando se ejecuta??  
}))
```

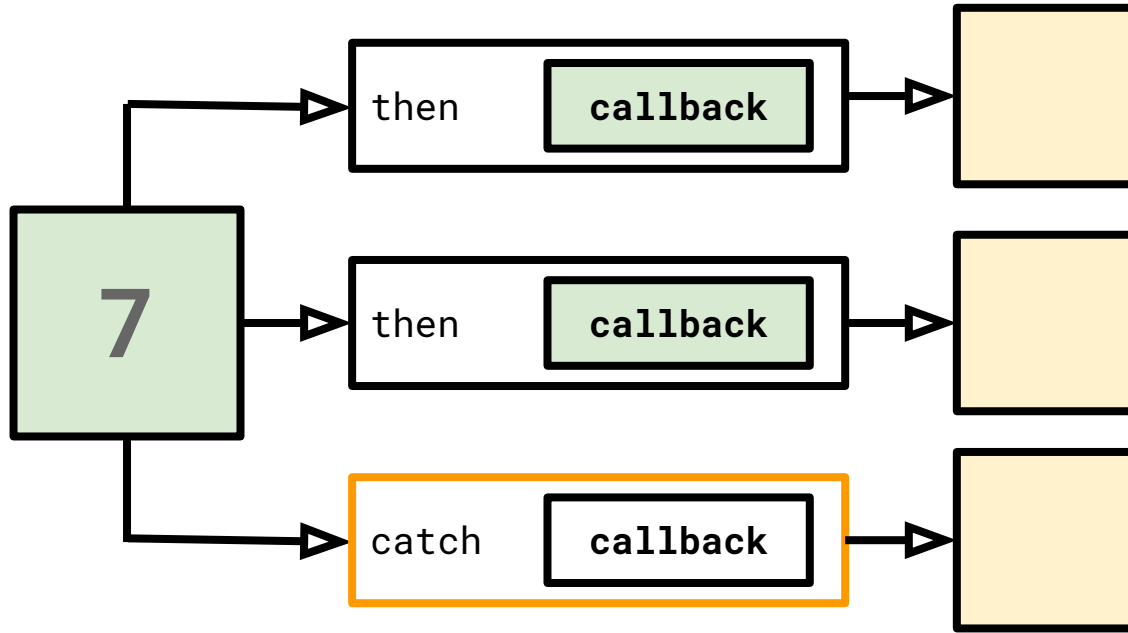
```
const promise = new Promise((resolve, reject) => {  
  setTimeout(() => resolve(42), 1000)  
}))
```

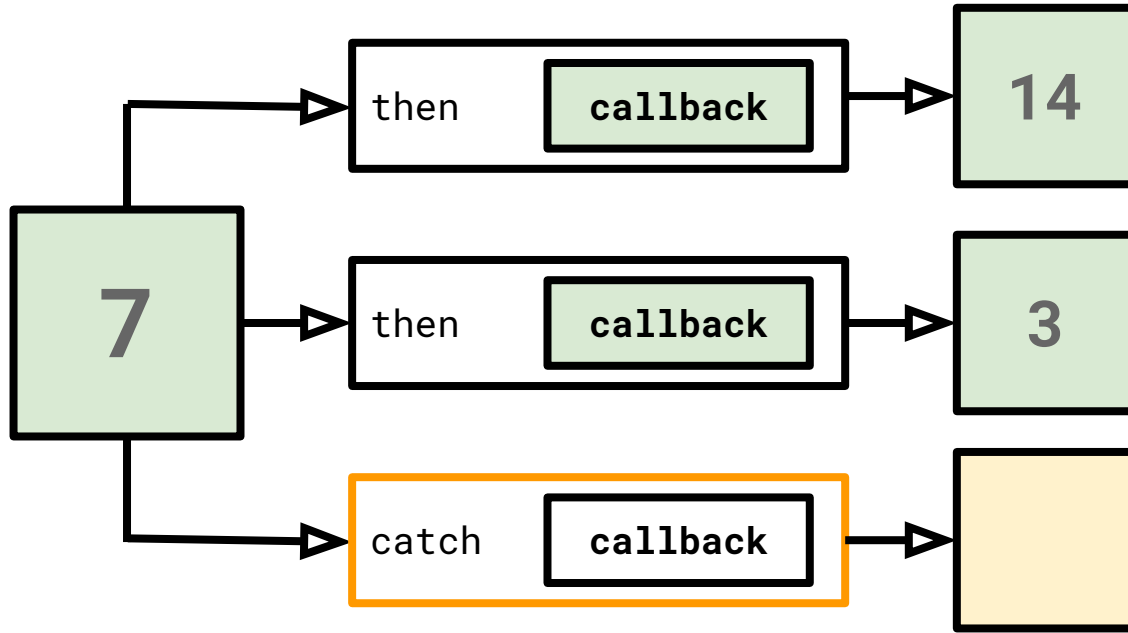
```
const promise2 = promise.then((value) => {  
  return value * 2  
}))
```

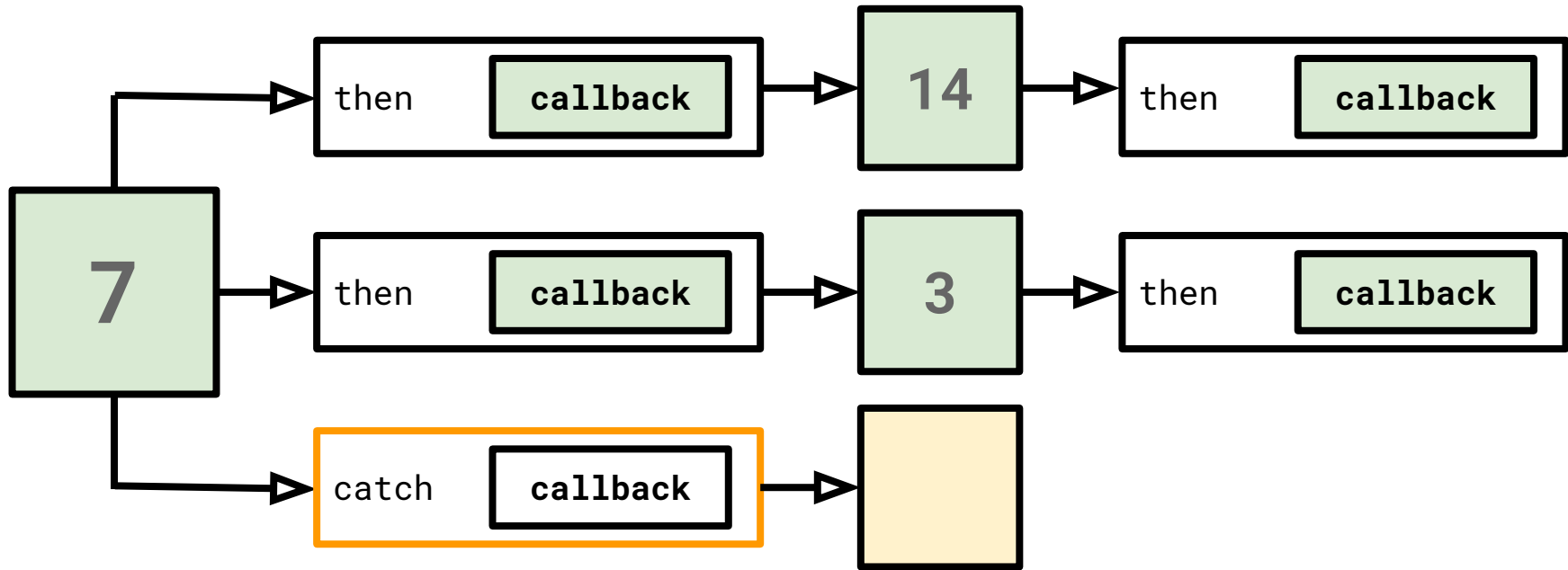
```
const promise3 = promise.then((value2) => {  
  console.log(value2) // ???  
}))
```

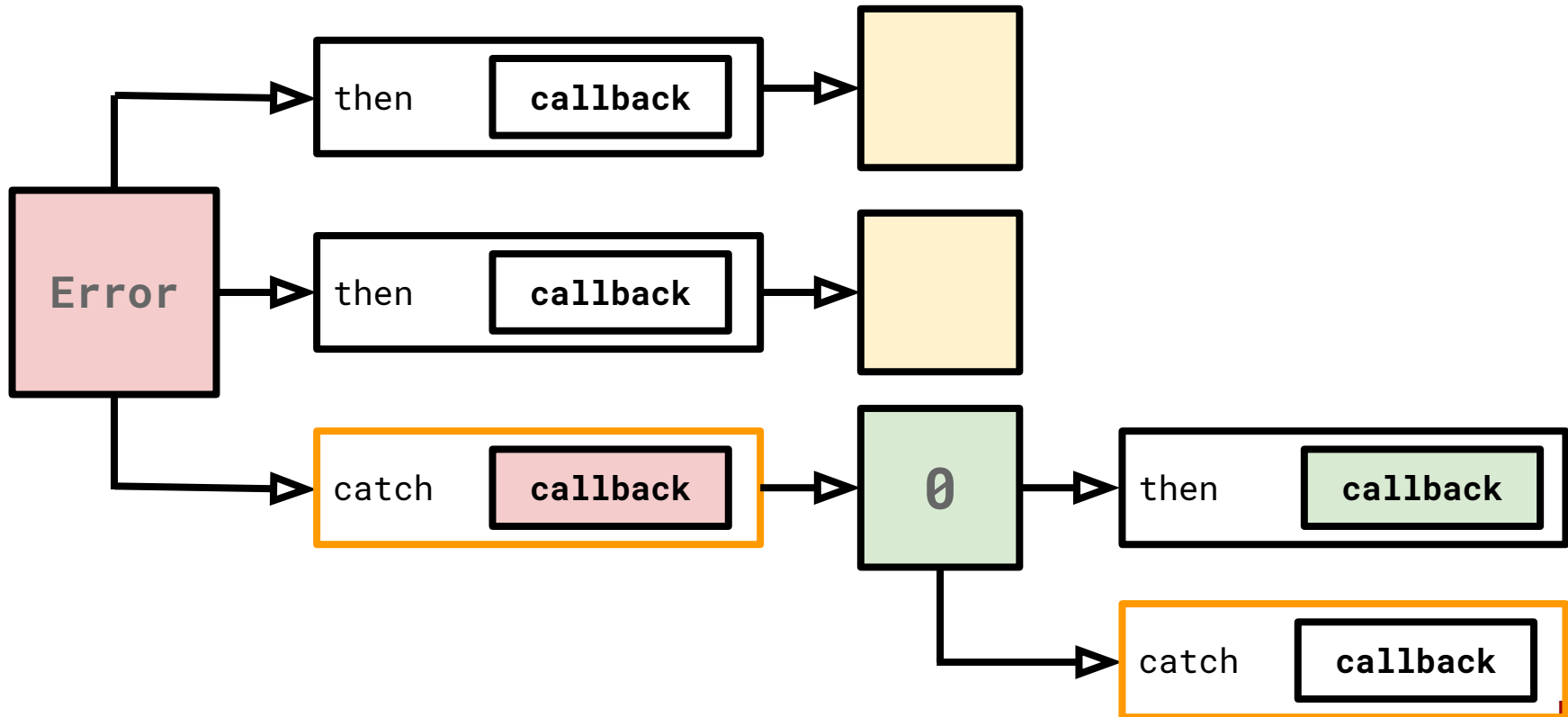













```
const promise = new Promise((resolve, reject) => {  
  setTimeout(() => reject(42), 1000)  
})
```

```
const promise2 = promise.then((value) => {  
  return value * 2  
})
```

```
const promise3 = promise.catch((error) => {  
  console.error(error)  
  return 0  
})
```

```
// (this is the one giving errors in the console)  
promise2.then(value2 => console.log('A', value2)) // ???  
promise2.catch(error2 => console.error('B', error2)) // ???  
promise3.then(value3 => console.log('C', value3)) // ???  
promise3.catch(error2 => console.error('D', error2)) // ???
```

Promesas

- Hay **tres** maneras de crear una promesa
 - `Promise.resolve(value)`
 - `Promise.reject(error)`
 - `new Promise(...)`

Promesas

- `Promise.resolve(value)`
 - Crea una promesa **resuelta**
 - Los **callbacks** de **.then** se ejecutan **inmediatamente**

```
const p = Promise.resolve('ready');  
p.then(console.log);
```

```
const p = Promise.resolve('ready');  
p.catch(console.log);
```

Promesas

- `Promise.reject(error)`
 - Crea una promesa **rechazada**
 - Los **callbacks** de **`.catch`** se ejecutan **inmediatamente**

```
const p = Promise.reject(new Error('doomed from the start'));  
p.catch(console.log);
```

```
const p = Promise.reject(new Error('doomed from the start'));  
p.then(console.log);
```

Promesas

- `new Promise(callback)`
 - Crea una promesa **pendiente**
 - El callback se ejecuta **con delay 0**
 - **callback** recibe dos parámetros
 - **resolve**: callback de resolución
 - **reject**: callback de rechazo

```
const p = new Promise((resolve, reject) => {  
  if (Math.random() < 0.5) resolve('good to go!')  
  else reject(new Error('bad luck'))  
}))
```

```
p.then(console.log)  
p.catch(console.log)
```

```
console.log('antes o después?') // ???
```


Promesas

- `.then(resolveCallback, [rejectCallback])`
 - Recibe uno (o dos) callbacks como parámetros
 - **resolveCallback**: recibe el valor de resolución
 - **rejectCallback**: recibe el error de rechazo

Promesas

- Una promesa se considera **rechazada** si **se levanta una excepción** durante la ejecución...
 - ...de su *callback de resolución*
 - ...o de su *callback de rechazo*

```
const p = new Promise((resolve, reject) => {  
  throw new Error('Oh, noes!')  
})  
  
p.then(  
  value => console.log(`Resuelta con: ${value}`),  
  error=> console.error(`Rechazada con: ${error}`),  
)
```

Promesas

- Una promesa se considera **rechazada** si **se levanta una excepción** durante la ejecución...
 - ...de su *callback de resolución*
 - ...o de su *callback de rechazo*

```
const p = Promise.resolve(true)
p.then(
  v => { throw new Error('rejected!') },
  err => console.error(err)
)
```

```
const p1 = new Promise(  
  resolve => setTimeout(() => resolve(1), 1000)  
);
```

```
const p2 = p1.then(v => v + 1);  
const p3 = p2.then(v => v + 1);  
const p4 = p3.then(v => v + 1);
```

```
p4.then(console.log); // ???
```

```
const p1 = new Promise(  
  resolve => setTimeout(() => resolve(1), 1000)  
);
```

```
p1.then(v => v + 1)  
  .then(v => v + 1)  
  .then(v => v + 1)  
  .then(console.log);
```

Promesas

- Si tenemos una promesa **A**
- Llamamos a **A.then(callback)** y nos devuelve la promesa **B**
- La promesa **B** se **resolverá** con el **valor de retorno** de **callback**

Promesas

- Si tenemos una promesa **A**
- Llamamos a **A.then(callback)** y nos devuelve la promesa **B**
- La promesa **B** se **resolverá** con el **valor de retorno** de **callback**
- Pero... ¿Y si **callback** devuelve otra promesa **C**?

```
const b = a.then(() => {  
  const c = new Promise(  
    resolve => setTimeout(() => resolve(1), 1000)  
  );  
  return c;  
});
```

```
b.then(console.log);
```

Promesas

- En ese caso, **B** se convierte en **un reflejo** de **C**
 - **B** sigue *pending* mientras **C** siga *pending*
 - Si **C** se resuelve, **B** se resuelve **con el mismo valor**
 - Si **C** se rechaza, **B** se rechaza **con el mismo error**

```
function futureValue(n) {  
  return new Promise(  
    resolve => setTimeout(() => resolve(n), 1000)  
  )  
}
```

```
futureValue(1)  
  .then(v => futureValue(v + 1))  
  .then(v => futureValue(v + 1))  
  .then(console.log) // ???
```

Promesas

- En una cadena de promesas
 - Los errores se **propagan hacia abajo**
 - Si se captura el error, la cadena **se resuelve con normalidad** a partir de ese punto
- Facilita el manejo de errores en procesos asíncronos

```
const p1 = new Promise((resolve, reject) => {  
  throw new Error('Oh, noes!')  
}))
```

p1

```
.then(() => console.log('1...'))  
.then(() => console.log('2...'))  
.then(() => console.log('3...'))  
.catch(() => console.log('Something bad happened'))  
.then(() => console.log('Everything under control'))
```

Promesas

- `Promise.all([prom1, prom2, prom3, ...])`
 - Devuelve **una nueva promesa**
 - Se resuelve cuando se **hayan resuelto todas** las promesas
 - **Valor de resolución:** valores de resolución de cada promesa
 - Si **una promesa es rechazada**, la promesa devuelta **se rechaza con el mismo error**

```
function futureValue(n, ms) {  
  return new Promise(resolve => setTimeout(() => resolve(n), ms));  
}
```

```
const p = Promise.all([  
  futureValue(1, 100),  
  futureValue(2, 200),  
  futureValue(3, 300),  
  futureValue(4, 400),  
]);
```

```
p.then(console.log); // [ 1, 2, 3, 4 ]
```



```
function futureValue(n, ms) {  
  return new Promise(resolve => setTimeout(() => resolve(n), ms));  
}
```

```
function futureFail(msg, ms) {  
  return new Promise(  
    (resolve, reject) => setTimeout(() => reject(msg), ms)  
  );  
}
```

```
const p = Promise.all([  
  futureValue(1, 100),  
  futureValue(2, 200),  
  futureFail('Bad luck', 100),  
]);
```

```
p.catch(console.log); // Bad luck
```

Promesas

- `Promise.race([prom1, prom2, prom3, ...])`
 - Devuelve **una nueva promesa**
 - **Refleja** el valor de la **primera promesa** que se **resuelva** o **rechace**

```
function futureValue(n, ms) {  
  return new Promise(resolve => setTimeout(() => resolve(n), ms));  
}
```

```
function futureFail(msg, ms) {  
  return new Promise(  
    (resolve, reject) => setTimeout(() => reject(msg), ms)  
  );  
}
```

```
const p = Promise.race([  
  futureValue(1, 100),  
  futureValue(2, 200),  
  futureFail('Bad luck', 200),  
]);
```

```
p.then(console.log, console.log); // 1
```

```
function futureValue(n, ms) {  
  return new Promise(resolve => setTimeout(() => resolve(n), ms));  
}
```

```
function futureFail(msg, ms) {  
  return new Promise(  
    (resolve, reject) => setTimeout(() => reject(msg), ms)  
  );  
}
```

```
const p = Promise.race([  
  futureValue(1, 100),  
  futureValue(2, 200),  
  futureFail('Bad luck', 50),  
]);
```

```
p.then(console.log, console.log); // Bad luck
```

Ejercicio: Promesas

- Implementa *mapPromise(fn, promisesOrValues)*
 - Aplica **fn** al *valor* de cada promesa de la lista
 - En paralelo
 - Devuelve una **promesa**
 - Que se resuelve a una lista de **valores**

Ejercicio: Promesas

- Implementa *mapSeriesPromise(fn, promisesOrValues)*
 - Sejemante a *mapPromise*
 - Pero **la iteración sucede en serie**

Ejercicio: Promesas

- Implementa *reducePromise(fn, init, promisesOrValues)*
 - Semejante a *reduce*, pero sobre promesas
 - **Iteración en serie**

Ejercicio: Promesas

```
reducePromise(  
  (acc, el) => futureValue(acc + i, 100),  
  [0, 1, 2, 3, futureValue(4), 5, 6, 7, 8, 9],  
  0  
)  
  .then(console.log); // 45
```