

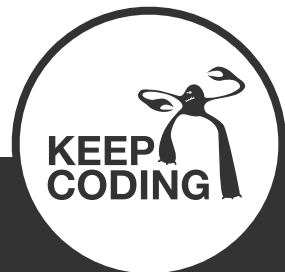
# Node.js

# Javascript II



# Javier Miguel

- CTO & Freelance Developer
- Email: [jamg44@gmail.com](mailto:jamg44@gmail.com)
- Twitter: [@javermiguelg](https://twitter.com/javermiguelg)

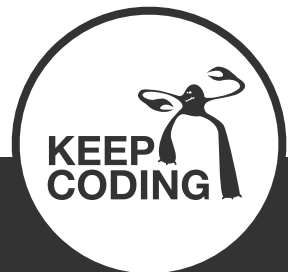


# ■ Immediately-Invoked Function Expression (IIFE)

Es una función que se crea como expression no como un statement.

```
( function() { /* code */ }() );
```

Se usa comúnmente para dar privacidad, no ensuciar el espacio global y retener valores (closures).



# ■ Immediately-Invoked Function Expression (IIFE)

```
var elems = document.getElementsByTagName( 'a' );

for ( var i = 0; i < elems.length; i++ ) {

    elems[ i ].addEventListener( 'click', function(e){
        e.preventDefault();
        alert( 'I am link #' + i );
    }, 'false' );

}
```

El link siempre reportará el último valor de i, no el que tenía cuando se preparó.



# ■ Immediately-Invoked Function Expression (IIFE)

```
var elems = document.getElementsByTagName( 'a' );

for ( var i = 0; i < elems.length; i++ ) {

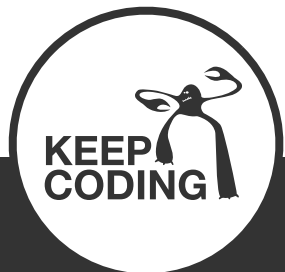
    (function( lockedInIndex ){

        elems[ i ].addEventListener( 'click', function(e){
            e.preventDefault();
            alert( 'I am link #' + lockedInIndex );
        }, 'false' );

    })( i );

}
```

El link reportará el valor de i en el momento de recorrerlo.



# ■ Immediately-Invoked Function Expression (IIFE)

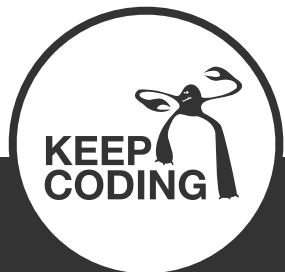
Entre estas dos formas de construirlas no hay diferencia:

```
(function() {  
    var foo = 'bar';  
})();
```

```
(function() {  
    var foo = 'bar';  
})();
```

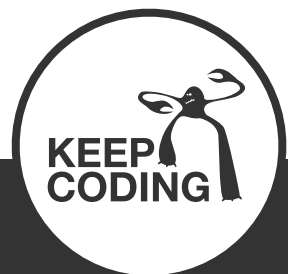
Douglas Crockford prefiere la segunda. A la primera la llama 'Dog balls' ;-)

<https://www.youtube.com/watch?v=eGArABpLy0k>





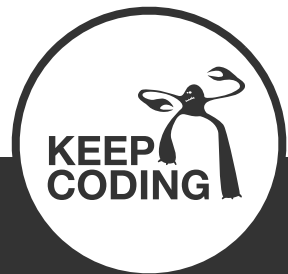
■ this



# ■ this

La palabra clave this trata de representar a quien llama a nuestra función como método, no donde está definida.

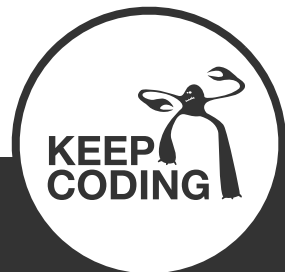
Por lo general, su valor hace referencia **al objeto propietario** de la función que la está invocando o en su defecto, al objeto donde dicha función es un método.





# ■ this

```
var persona = {  
  name: 'Luis',  
  surname: 'Gomez',  
  fullname: function() {  
    console.log(this.name + ' ' + this.surname);  
  }  
};  
  
persona.fullname(); // Luis Gomez
```

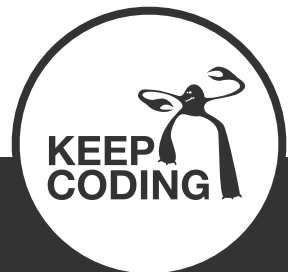


# ■ this

De forma general, cuando se usa en algo **distinto a un método** su valor es el contexto global (o *undefined* si estamos en modo estricto).

```
console.log(this); // window en un browser, global en node
```

```
function pinta(){  
  console.log(this); // window en un browser, global en node  
}
```



# ■ this

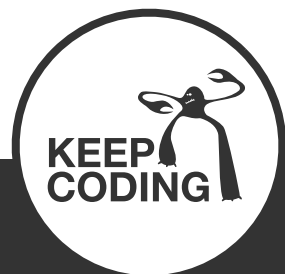
Averigua porque...

```
var persona = {  
  name: 'Luis',  
  surname: 'Gomez',  
  fullname: function() {  
    console.log(this.name + ' ' + this.surname);  
  }  
};
```

```
persona.fullname(); // Luis Gomez
```

```
setTimeout(persona.fullname, 1000); // undefined undefined
```

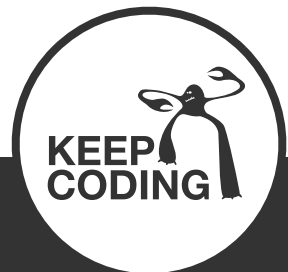
```
// pista: quien está invocando realmente la función fullName?
```



# ■ this

Como manejarlo? Le asignamos this con *bind*

```
var persona = {  
  name: 'Luis',  
  surname: 'Gomez',  
  fullname: function() {  
    console.log(this.name + ' ' + this.surname);  
  }  
};  
  
setTimeout(persona.fullname.bind(persona), 1000); // Luis Gomez
```



# ■ this

Otras formas de llamar a un método especificando cual será su 'this'

```
funcion.call(thisArg[, arg1[, arg2[, ...]]])
```

```
persona.iniciales.call(programa, 2, true);
```

```
funcion.apply(thisArg[, argsArray])
```

```
persona.iniciales.apply(programa, [2, false]);
```

La diferencia es:

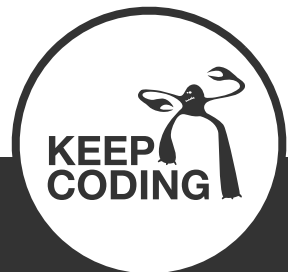
- en **c**all hay que poner todos los argumentos separados por **c**omas
- en **a**pply se le pasan como un **a**rray



# ■ this - uso en constructores

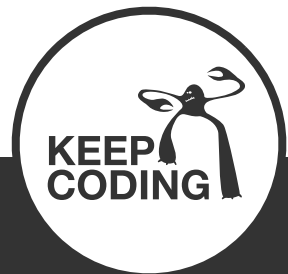
Cuando usamos this en un constructor de objetos, *este* apuntará al objeto retornado por el constructor.

```
function Coche() {  
  this.ruedas = 4;  
  this.logRuedas = function() {  
    console.log('tiene ' + this.ruedas);  
  }  
}  
  
var coche = new Coche();  
console.log(coche.ruedas); // 4  
coche.logRuedas(); // tiene 4  
setTimeout(coche.logRuedas, 1000); // tiene undefined
```





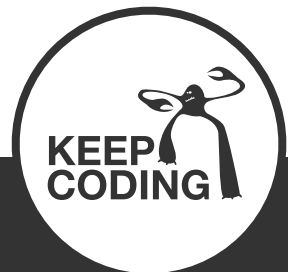
# Closures



# ■ Closures

Un closure se construye con una función (A) que devuelve otra (B).

La función devuelta (B), sigue manteniendo el acceso a todas las variables de la función que la creó (A).

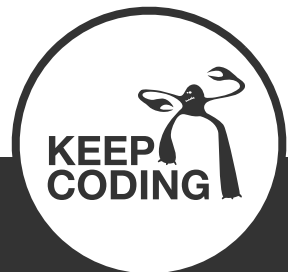




# ■ Closures

Ejemplo:

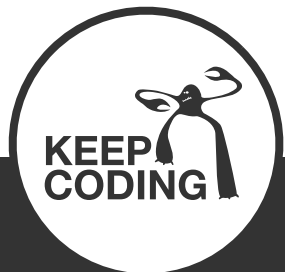
```
function creaClosure(valor) {  
  return function() {  
    return valor;  
  }  
}
```



# Closures

Algo más elaborado:

```
function creaAgente(nombre) {  
  var edad = 0;  
  return {  
    ponNombre: function(nuevoNombre) {  
      nombre = nuevoNombre;  
    },  
    leeNombre: function() {  
      return nombre;  
    },  
    ponEdad: function(nuevaEdad) {  
      edad = nuevaEdad;  
    },  
    leeEdad: function() {  
      return edad;  
    }  
  }  
}
```





# Prototipos



# ■ Prototype chain

Casi todo en Javascript es un objeto. Cada objeto tiene una propiedad interna llamada prototype que apunta a otro objeto.

Su objeto prototipo tiene a su vez una propiedad prototype que apunta a otro objeto, y así sucesivamente.

A esto se le llama cadena de prototipos.

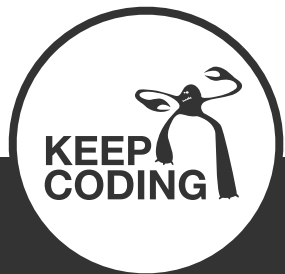
Si sigues la cadena en algún momento llegarás al objeto Object, cuyo prototipo es null.



# ■ Prototipos

Cuando pides una propiedad a un objeto el interprete mira a ver si la tiene ese objeto. Si no la encuentra mira a ver si la tiene su prototipo, y así hasta llegar al final de la cadena.

Esto nos permitiría hacer algo parecido a clases e implementar herencia.

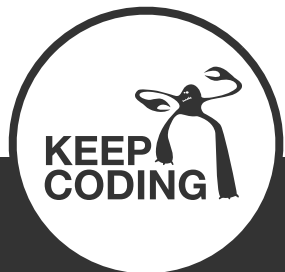


# ■ Prototipos

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

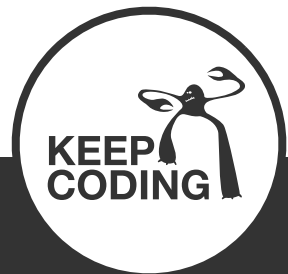
```
Persona.prototype.saluda = function() {  
    console.log("Hola, me llamo " + this.name);  
};
```

Esto hará que todas las personas sepan saludar, ¡incluso las que ya estén creadas!





# ■ Herencia de Prototipos

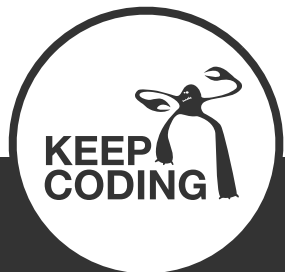


# ■ Prototipos - herencia

```
function Agente(name) {  
  Persona.call(this, name);  
  // heredamos el constructor  
  // Esto ejecutará el constructor de Persona sobre el this de Agente  
  // definiendo en el this de Agente sus propiedades.  
  // Es como llamar a "super" en otros lenguajes  
}  
  
// heredamos las propiedades de prototipo  
Agente.prototype = new Persona(); // heredaría name y saluda()
```

Así Agente hereda de Persona.

[ejemplos/prototipos.js](#)







# ■ Herencia múltiple



# ■ Herencia múltiple

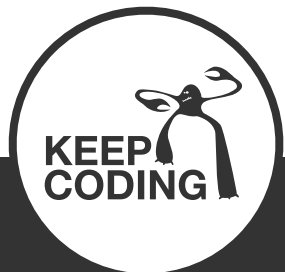
Podemos copiar las propiedades de un objeto a otro. Esto se suele llamar extender un objeto con otro.

Para esto podemos usar `Object.assign`:

```
Object.assign(objetoDestino, objetoFuente);
```

Documentación:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)

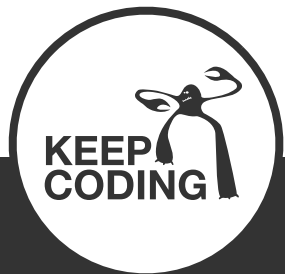


# ■ Herencia múltiple

Una forma de conseguir herencia múltiple es usar el patrón mixin:

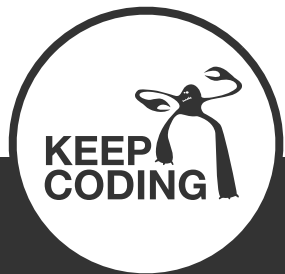
- Para conseguirlo heredamos del principal (ya lo hemos visto antes)
- Luego extendemos el prototipo del heredado con los otros:



```
Object.assign(Agente.prototype, matrixMixin);
```





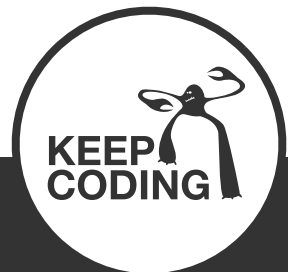
■ Imperativo, funcional, OOP, declarativo







Imperativo: El código se estructura en el orden de ejecución, usando sub-rutinas para conseguir reutilización. Por ejemplo como en C, Pascal, PHP.

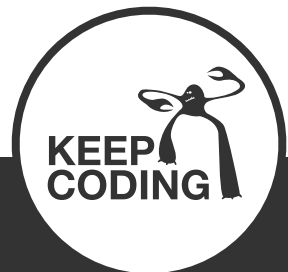
Funcional: Se hacen funciones que se aplican a los datos. Por ejemplo, para procesar un array en vez de hacer un bucle, hacemos una función que evalúa un elemento y se llama a si misma con el siguiente. Ejemplos: Scheme, Erlang, Haskell, Lisp





Orientado a objetos (OOP): Se definen objetos con propiedades y métodos similares a los que intervienen en el problema que se quiere solucionar. Por ejemplo como en Java, C#

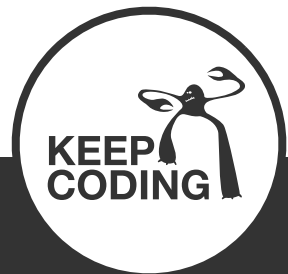
Declarativo: Se le dice al ordenador que es lo que hay que hacer, pero no los detalles de como hacerlo. Por ejemplo SQL





# ■ Ejercicio

Versión módulos

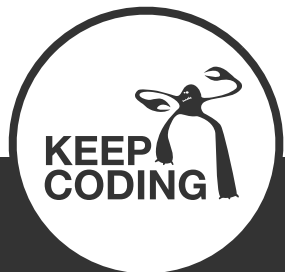


# ■ Ejercicio - versión de módulos

Modificar la función **versionModulo** para que consiga la versión de todos los módulos instalados en mi proyecto.

Más ingredientes:

- fs.readdir ([https://nodejs.org/api/fs.html#fs\\_fs\\_readdir\\_path\\_callback](https://nodejs.org/api/fs.html#fs_fs_readdir_path_callback))
- async.concat (<https://github.com/caolan/async#concat>)
- async common pitfalls <https://github.com/caolan/async#common-pitfalls-stackoverflow>
- fs.statSync ([https://nodejs.org/api/fs.html#fs\\_fs\\_statsync\\_path](https://nodejs.org/api/fs.html#fs_fs_statsync_path))





# ■ Ejercicio - versión de un módulo

Recibirá un callback. Debe devolver un posible error y un array con los nombres y versiones de los módulos que encuentre.

La probaremos con un código como este:

```
versionModulos (function(err, moduleArr) {  
    if (err) {  
        console.error('Hubo un error: ', err);  
        return;  
    }  
    console.log('Los módulos son:', moduleArr);  
});
```

