

Curso Intermedio de Programación Orientada a Objetos en JavaScript

static: atributos y métodos estáticos en JavaScript.

```
class Patito {  
    static sonidito = "cuak!";  
}  
  
console.log(Patito.sonidito);  
// cuak!
```

```
class Patito {  
    static hacerSonidito() {  
        return "cuak!";  
    }  
}  
  
console.log(Patito.hacerSonidito());  
// cuak!
```

Los métodos y atributos estáticos son llamados sin instanciar su clase y no pueden ser llamados mediante una instancia de clase. Son habitualmente utilizados para crear funciones para una aplicación.

```
Metodos estaticos de SuperPrototipo Object  
  
Object.keys(obj) // Devuelve un array con el nombre de las keys de nuestro objeto  
//[ "name", "email", "age" ]  
  
Object.getOwnPropertyNames(obj) // Devuelve un array con las propiedades de nuestro objeto  
//[ "name", "email", "age" ]  
  
Object.entries(obj) // Devuelve un array de arrays que contiene las keys y sus valores  
/*  
  0: [ "name", "juanito" ]  
  1: [ "email", "juanito@alimañana.com" ]  
  2: [ "age", 20 ]  
*/  
  
Object.getOwnPropertyDescriptors(obj) // ✨ Devuelve las propiedades de nuestro objeto. Esta es  
la forma de JavaScript para limitar el acceso o modificacion de nuestros atributos u objetos  
  
/*  
  name: {  
    value: "Juanito",  
    writable: true,  
    configurable: true,  
    enumerable: true  
  }  
*/
```

El método **bind()** crea una nueva función, que cuando es llamada, asigna a su operador **this** el valor entregado, con una secuencia de argumentos dados precediendo a cualquiera entregados cuando la función es llamada.

Ejemplo:

```
> juan
< ▼ {name: 'Juanito', age: 18, approvedCourses: Array(1), addCourse: f} ⓘ
  ► addCourse: f addCourse(newCourse)
    age: 18
  ► approvedCourses: (2) ['Curso 1', 'Curso 2']
    name: "Juanito"
  ► [[Prototype]]: Object

> Object.entries(juan)[3];
< ► (2) ['addCourse', f]

> Object.entries(juan)[3][1];
< f addCourse(newCourse) {
  console.log("this", this);
  console.log("this.approvedCourses", this.approvedCourses);
  this.approvedCourses.push(newCourse);
}

> Object.entries(juan)[3][1].bind(juan)("Curso 2");
```

Si no fuera por el método **bind()**, **this** en **Object.entries(juan)[3][1]** haría referencia al array **['addCourse', f]** y por lo tanto no existiría la propiedad **approvedCourses** y no funcionaría el método **push()**.

El método estático **Object.defineProperty()** define una nueva propiedad sobre un objeto, o modifica una ya existente, y devuelve el objeto modificado.

Este método nos permite modificar el comportamiento por defecto de las propiedades. Es decir, nos permite definir una propiedad como no enumerable, no modificable o incluso evitar que pueda ser eliminada del objeto.

```
Object.defineProperty(juan, "pruebaNasa", {
  value: "alien",
  writable: false, //No se puede editar el valor de la propiedad
  enumerable: false, //No aparece la propiedad en Object.keys(juan) ni el for(let keys in juan), pero sí en Object.getOwnPropertyNames(juan)
  configurable: false, // delete juan.pruebaNasa no funciona, no puedes borrar la propiedad.
});
```

El método **Object.getOwnPropertyDescriptors()** regresa todos los descriptores de propiedad propios de un objeto dado.

IMAGEN ABAJO

```
> Object.getOwnPropertyDescriptors(juan);
< ▼ {name: {...}, age: {...}, approvedCourses: {...}, addCourse: {...}, pruebaNasa: {...}, ...} ⓘ
  ► addCourse: {writable: false, enumerable: true, configurable: false, value: f}
  ► age: {value: 18, writable: false, enumerable: true, configurable: false}
  ► approvedCourses: {value: Array(2), writable: false, enumerable: true, configurable: false}
  ► editor: {value: 'VSCode', writable: false, enumerable: true, configurable: false}
  ► name: {value: 'Juanito', writable: false, enumerable: true, configurable: false}
  ► navigator: {value: 'Chrome', writable: false, enumerable: false, configurable: false}
  ► pruebaNasa: {value: 'alien', writable: false, enumerable: false, configurable: false}
  ► terminal: {value: 'WSL', writable: false, enumerable: true, configurable: false}
  ► [[Prototype]]: Object
```

Un **descriptor de propiedad** es un registro con algunos de los siguientes atributos:

- **value:** El valor asociado con la propiedad (solo descriptores de datos).
- **writable:** true si y solo si el valor asociado con la propiedad puede ser cambiado (solo descriptores de datos).
- **get:** Un función que sirve como un getter para la propiedad, o undefined si no hay getter (solo descriptores de acceso).
- **set:** Una función que sirve como un setter para la propiedad, o undefined si no hay setter (solo descriptores de acceso).
- **configurable:** true si y solo si el tipo de este descriptor de propiedad puede ser cambiado y si la propiedad puede ser borrada de el objeto correspondiente.
- **enumerable:** true si y solo si esta propiedad aparece durante la enumeración de las propiedades en el objeto correspondiente. En caso de false, no aparece la propiedad en `Object.keys(juan)` ni el `for(let keys in juan)`, pero sí en `Object.getOwnPropertyNames(juan)`

Object.seal(obj) hace que las propiedades no se puedan borrar {configurable: false}.

Object.freeze(obj) hace que las propiedades no se puedan borrar ni editar {configurable: false, writable: false}

Con **Object.isSealed** podemos comprobar si todas las propiedades de un objeto están bloqueadas a que sean eliminadas. Nos devolverán un booleano. Ejemplo:

Object.isSealed(juan);

Con **Object.isFrozen(juan):** Lo mismo que `Object.isSealed` pero tambien comprobar que no puedan ser editadas las propiedades. {writable: false}

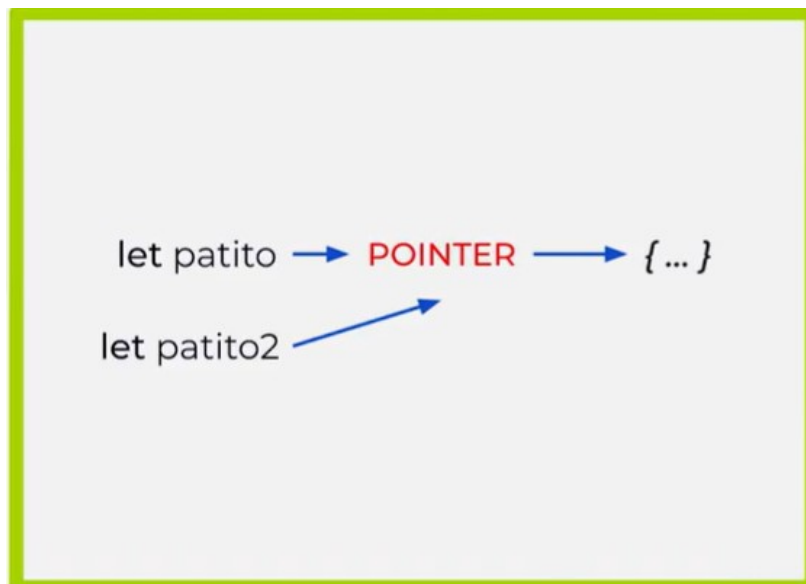
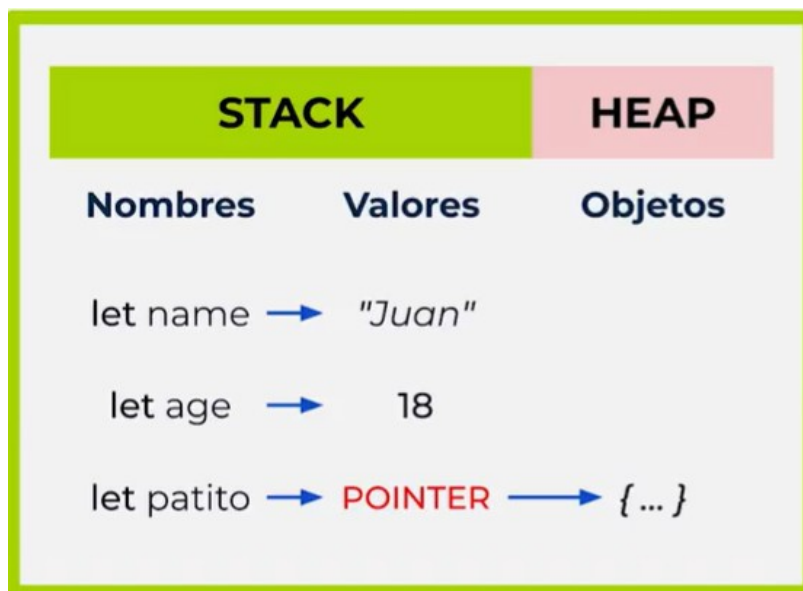
Lo que ocurre si la propiedad es false

	Enumerable	Writable	Configurable
Listar con <code>Object.keys()</code>	✗	✓	✓
Listar <code>getOwnPropertyNames</code>	✓	✓	✓
Modificar value	✓	✗	✓
Eliminar propiedad del objeto	✓	✓	✗

Los objetos son referencia a un espacio en memoria. Cuando copiamos un objeto en realidad copiamos su referencia en la memoria. Por lo que al modificar la copia del objeto también modificamos el objeto original.

- Las **variables** son referencias a un espacio en memoria.
- Los **navegadores web** usan dos tipos de memorias: Stack y Heap.
- La **memoria Stack** es muy rápida, pero sin tanto espacio. Aquí se guardan los valores primitivos (booleanos, strings, números...).
- La **memoria Heap** es más lenta, pero permite guardar enormes cantidades de información (son como los tornados: grandes, lentos y desordenados). En esta memoria guardamos los valores de los objetos ({...}).

Entender cómo funciona la memoria en JavaScript no solo será útil para aprender POO, sino también para programación funcional.



El método **Object.assign()** copia todas las propiedades enumerables de uno o más objetos fuente a un objeto destino. Devuelve el objeto destino.

Las propiedades en el objeto destino serán sobrescritas por las propiedades en las fuentes si tienen la misma clave. Propiedades posteriores de las fuentes podrán sobrescribir las anteriores.

El método `Object.assign()` copia sólo las propiedades **enumerables** y propias del objeto origen a un objeto destino.

Advertencia para clonado profundo: Para un clonado profundo, necesitamos usar otra alternativa ya que `Object.assign()` copia valores de propiedades. Si el valor en la fuente es una referencia a un objeto, solo se copia la referencia en sí, como valor de la propiedad.

```
1 const target = { a: 1, b: 2 };
2 const source = { b: 4, c: 5 };
3
4 const returnedTarget = Object.assign(target, source);
5
6 console.log(target);
7 // Expected output: Object { a: 1, b: 4, c: 5 }
8
9 console.log(returnedTarget === target);
10 // Expected output: true
```

El método **Object.create()** crea un objeto nuevo, utilizando un objeto existente como el prototipo del nuevo objeto creado. Esto es para herencia simple.

```
> obj1
< ▾ {a: 1, b: 'b', c: {...}} ⓘ
  a: 1
  b: "b"
  ▶ c: {d: 'd', e: 'e'}
  ▶ [[Prototype]]: Object

> const anotherObj = Object.create(obj1);
< undefined

> anotherObj
< ▾ {} ⓘ
  ▾ [[Prototype]]: Object Herencia
    a: 1
    b: "b"
    ▶ c: {d: 'd', e: 'e'}
    ▶ [[Prototype]]: Object
```

¿Qué es la recursividad?

Es el acto de una función llamándose a sí misma. La recursión es utilizada para resolver problemas que contienen subproblemas más pequeños. Una función recursiva puede recibir 2 entradas: un caso base (finaliza la recursión) o un caso recursivo (continúa la recursión).

```
function recursiva(merito) {  
  console.log(merito);  
  if (merito < 5) {  
    return recursiva(merito + 1);  
  } else {  
    return 5;  
  }  
}
```

Factory pattern (o fábrica de objeto) y **RORO** (Recibir un Objeto, Retornar un Objeto) son dos patrones que nos ayudan a crear moldes de objetos a partir de funciones. Con ello ya no sería necesario utilizar objetos literales ni deep copy con recursividad.

En JavaScript no tenemos keywords para indicar que un atributo es **privado** o **público** a diferencia de otros lenguajes de programación. Sin embargo, podemos aplicar ciertas técnicas y métodos para lograrlo.

Podemos evitar que el usuario modifique o elimine métodos/propiedades y dar así mejor seguridad a estos. Con **Object.defineProperty** podemos hacer las configuraciones respectivas para evitar lo mencionado.

La desventaja de protegerlos es que no nos permitiría trabajar con el **polimorfismo** (uno de los pilares de POO).

Por ejemplo, si tuviéramos dos métodos que deseamos proteger de sobreescritura (polimorfismo) y de ser eliminados de su objeto, podemos protegerlos de la siguiente manera.

```
Object.defineProperty(publicAtributos, "readName", { // 👉👉  
  writable: false,  
  configurable: false,  
});  
Object.defineProperty(publicAtributos, "changeName", { // 👉👉  
  writable: false,  
  configurable: false,  
});
```

Aquí vemos que ambos métodos pertenecen a un objeto llamado "publicAtributos". Y dicho objeto puede inclusive estar dentro del objeto principal.

La sintaxis **get** vincula la propiedad de un objeto con una función que se llamará cuando se busque esa propiedad.

La sintaxis **set** vincula la propiedad de un objeto con una función que se llamará cuando se intente hacer una asignación a esa propiedad.

```
get name() {  
    return privateAtributos["_name"];  
},  
set name(newName) { // 👉 👉  
    privateAtributos["_name"] = newName;  
}
```


```
const juan = createStudent({ email: "juanito@frijoles.co", name: "Juanito" });  
  
console.log(juan.name); // Se ejecuta el GETTER  
juan.name = "Rigoberto"; // Se ejecuta el SETTER  
console.log(juan.name);
```

Si aplicamos **Object.getOwnPropertyDescriptors** sobre nuestro objeto juan para visualizar la accesibilidad de sus atributos: el atributo name no tendrá las propiedades **value** y **writable** como tal, en vez de eso nos aparecerán las funciones get y set.

Default levels ▾ No Issues

- ▶ **approvedCourses**: {value: Array(0), writable:...
- ▶ **email**: {value: "juanito@frijoles.co", writab...
- ▶ **learningPaths**: {value: Array(0), writable: t...
- ▼ **name**:
 - configurable: true
 - enumerable: true
 - ▶ **get**: f name()
 - ▶ **set**: f name(newName)
 - ▶ **__proto__**: Object
- ▶ **socialMedia**: {value: {...}, writable: true
- ▶ **__proto__**: Object

>



El **duck typing** es la forma de programar donde identificamos los elementos por los métodos y atributos que tenga por dentro.

¿Cómo funciona el duck typing?

Se deben tener parámetros para saber diferenciar dos cosas, dos personas, dos elementos, etc. Si queremos determinar quién es quién, se debe mirar por sus atributos y métodos, aunque puede haber el caso en el que haya elementos parecidos que también se deben diferenciar (impostores), es cuando más detalle se debe poner en identificar qué los componen.

El nombre proviene de la frase: *“Si parece un pato y grazna como un pato, es un pato.”* En otras palabras, tiene que cumplir con ciertos métodos y atributos para considerarse alguna cosa.

Con **instanceof** podemos saber si un objeto es instancia de cierto prototipo. Esto nos devuelve true o false. Y es una forma que nos puede ayudar con el duck typing. Por ejemplo, podremos saber si le estamos pasando un objeto literal (elemento impostor en este caso) o una instancia de un prototipo como argumento a una función.

```
const escuelaWeb = new LearningPath({
  name: "Escuela de desarrollo web",
  courses: ["JS for beginners", "JS intermedio"]});

const escuelaData = new LearningPath({name: "Escuela de desarrollo web"});

const juan = new Student({
  email: "juanito@frijoles.co",
  name: "Juanito",
  learningPaths: [escuelaWeb, escuelaData]});

const mario = new Student({
  email: "mario@frijoles.co",
  name: "Mario",
  learningPaths: [escuelaWeb, {name: "Escuela del impostor", courses: []}]});
```

```
for (elemento of learningPaths) {
  //console.log({learningPaths, elemento, instanceof: elemento instanceof LearningPath});

  /*En el if se requiere envolver la expresión en parentesis porque el operador not(!) tiene
  mayor precedencia que el instanceof*/
  if(!(elemento instanceof LearningPath)) {
    console.warn("learningPath NO es un verdadero LearningPath");
    return;
  }
}
```