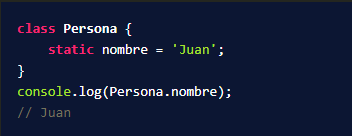
# **Static: atributos y métodos estáticos en JavaScript**

Para acceder a los **métodos**, **atributo** o **clases** de un prototipo hay que tener una instancia, pero esto no es necesario para los métodos estáticos.

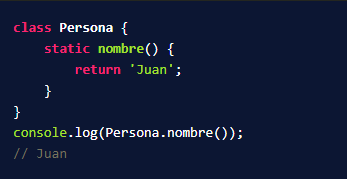
## **static**

Para crear atributos estáticos a los que podamos acceder sin crear un objeto, le ponemos la palabra reservada **static**.

## **Propiedades estáticas**



## **Métodos estáticos**



Cuando se llame por fuera, sin crear instancias, también obtendremos el ‘Juan’

## **Prototipo object**

Padre de todos los prototipos.

## **Object.keys()**

Nos devuelve un array con los **nombres clave** de los atributos del objeto.



## **Object.entries()**

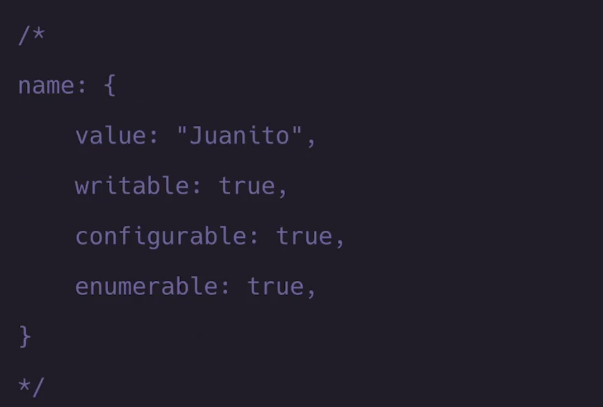
Nos devuelve un array con los **keys** y los **value** del objeto.



## **Object.getOwnPropertyDescriptors()**

Nos devuelve un objeto con las propiedades de nuestro objeto “writable, enumerable, configurable, value”.

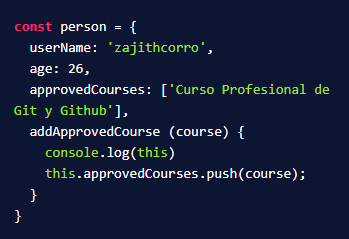
Esta es la forma que tiene javascript por dentro para limitar el acceso o la modificación de nuestros **atributos** u **objetos**.



# **Object.defineProperty**

Con esta propiedad se pueden definir nuevas propiedades a nuestro objeto. Así mismo, se puede configurar ciertas características de la propiedad tales como:

* **Configurable**: Esta indica si la propiedad puede ser borrada o eliminada.
* **Enumerable**: Indica si la propiedad puede ser mostrada en la enumeración de las mismas. Existen ciertos métodos que toman como referencia este valor para mostrar la propiedad.
* **Writable**: Esta indica si la propiedad puede ser modificada con el operador de asignación.





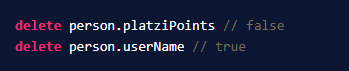
Si queremos modificar una propiedad que tienen **writable**: **false** no permitirá que su valor sea modificado.



**Object.keys** solo muestra las propiedades que tienen **enumerable**: **true**. A diferencia de **Object.getOwnPropertyNames** que muestra todas las propiedades.



Si queremos eliminar propiedad que tienen **configurable**: **false** no permitirá que sea borrada del objeto.



## **Object.freeze()**

Este método congela un objeto que sea pasado por parámetro. Es decir:

* Impide que se le agreguen nuevas propiedades.
* Impide que sean eliminas propiedades ya existentes.
* Impide que sus las propiedades internas (writable, enumerable y configurable) sean modificadas.



## **Object.seal()**

Este método sella un objeto que sea pasado por parámetro. Es decir:

* Impide que nuevas propiedades sean agregadas.
* Cambia en todas las propiedades **configurable**: **false**, con lo que impide que sean borradas.
* Las propiedades aún pueden ser modificadas, ya que **writable** está **true**.



# **Cómo funciona la memoria en JavaScript**

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, etc).
* **La memoria Heap** es más lenta, pero permite guardar enormes cantidades de información. 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.

## **Copiar objetos en JavaScript**

Cuando se realiza una copia de un objeto, lo que en realidad ocurre es que se copia la referencia a la memoria de ese objeto, es decir, el apuntador. Esto hace que, al modificar la copia, también se modifique el objeto original.

# **Shallow copy en JavaScript**

Shallow Copy se refiere a la forma de crear un nuevo objeto a partir de las propiedades de otro. Esta copia solo se hace a un nivel alto, no se hace con objetos dentro de objetos (nested objects), lo que provoca que la modificación de una de sus propiedades, modifique el objeto principal.

# **Qué es JSON.parse y JSON.stringify**

## **JSON.stringify()**

El método **JSON.stringify()** convierte un objeto o valor de JavaScript en una cadena JSON, reemplazando opcionalmente valores si se especifica una función de reemplazo u opcionalmente incluyendo solo las propiedades especificadas si se especifica una matriz de reemplazo.

* Los objetos **Boolean**, **Number**, and **String** se convierten a sus valores primitivos, de acuerdo con la conversión semántica tradicional.
* Si durante la conversión se encuentra un **undefined**, una **Function**, o un **Symbol** se omite (cuando se encuentra en un objeto) o se censura a **null** (cuando se encuentra en un array). JSON.stringify() puede devolver **undefined** cuando se pasan valores “puros” como **JSON.stringify(function(){})** o **JSON.stringify(undefined)**.
* Todas las propiedades que utilicen **Symbol** en los nombres de la clave se ignoran por completo, incluso si utilizan una función **replace**.
* Las instancias de **Date** implementan la función **toJSON()** devolviendo una cadena de texto (igual que **date.toISOString()**). Por lo que son tratadas como strings.
* Los números **Infinity** y **NaN**, así como el valor **null**, se consideran **null**.
* El resto de instancias de **Object** (incluyendo **Map**, **Set**, **WeakMap**, y **WeakSet**) sólo tendrán serializadas sus propiedades enumerables.

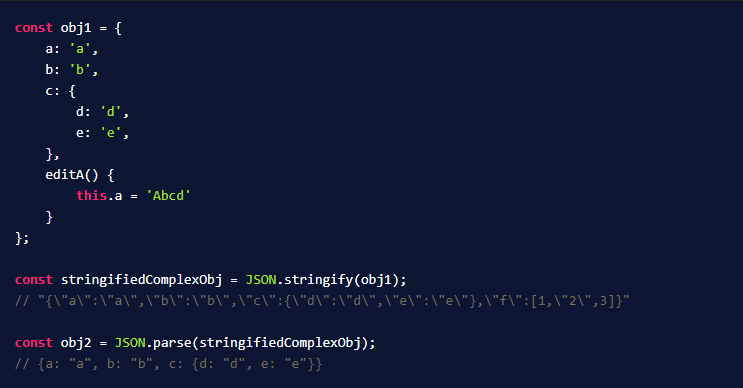
**JSON.stringify()** convierte un valor en notación JSON que lo representa.

## **JSON.parse()**

El método **JSON.parse()** analiza una cadena de texto (string) como JSON, transformando opcionalmente el valor producido por el análisis.

**Por qué JSON.parse(JSON.stringify()) es una mala práctica para clonar un objeto en JavaScript**.

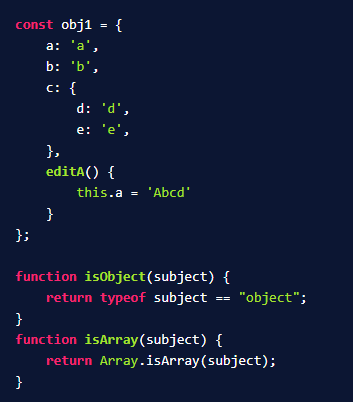
* Puedes perder tipos de datos.
* JavaScript no te avisara cuando pierdas algún tipo de dato al usar **JSON.stringify()**.
* Convierte tipos de datos no soportados en soportados, como **infinity** y **NaN** en **null**.
* Los tipos de datos **Date** serán parseados como strings, **no como Date**.
* No es tan rápido y eficiente.

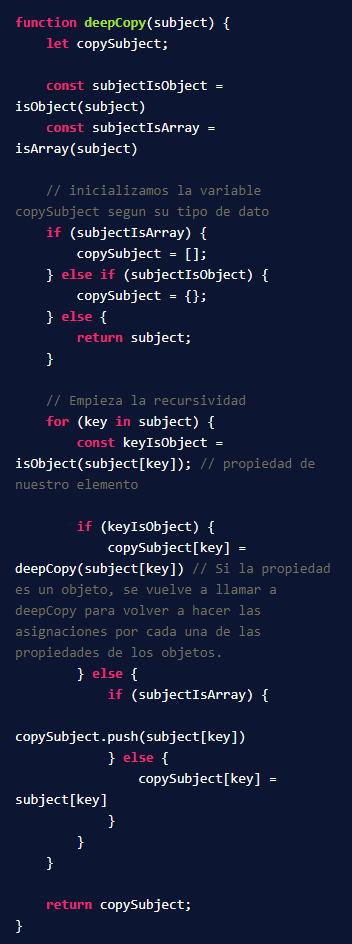


# **Deep copy con recursividad**

La recursividad consiste en funciones que se llaman a sí mismas, evitando el uso de bucles y otros iteradores.

Se dice que el objeto tiene un **Deep Copy** cuando cada propiedad del objeto **apunta a una copia separada**, incluso si la propiedad apunta a un objeto (valores de referencia). Se crea una copia separada del objeto de referencia para el objeto de destino. En caso de copia profunda, las propiedades de referencia del objeto de origen y las propiedades de referencia del objeto de destino **apuntan a diferentes ubicaciones de memoria**.





# **Abstracción con objetos literales y deep copy**

## **Object.isSealed()**

El método Object.isSealed() si el objeto está sellado.

Devuelve **true** si el objeto está sellado, de lo contrario devuelve **false**. Un objeto está sellado si **no es extensible** y **si todas sus propiedades no se pueden configurar** y por lo tanto no removibles (pero no necesariamente no modificables).

## **Object.isFrozen()**

Determina si un objeto está congelado.

Un objeto está congelado si y solo si no es extensible, todas sus propiedades son **no-configurables**, y todos los datos de sus propiedades **no tienen capacidad de escritura**.



# **Factory pattern y RORO**

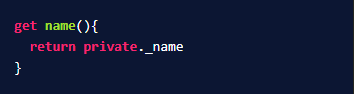
**RORO** (**R**ecibir un **O**bjecto, y **R**etornamos otro **O**bjecto.)

Es un patrón de código en el que se envía como parámetro a una función un **objeto** para devolver **otro objeto**, y el beneficio de esto principalmente es:

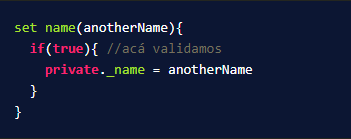
* Enviar parámetros de forma más descriptiva.
* Ignorar el orden en el que deben ser colocados los parámetros.

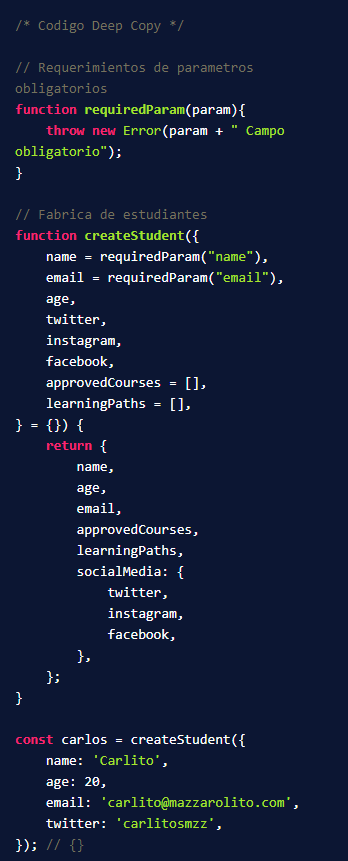
# **Getters y setters**

* **Getter**: Nos permite acceder al valor de una propiedad.



* **Setter**: Nos permite setear el valor a una propiedad.





# **Qué es duck typing**

Es la forma de programar donde identificamos a nuestros elementos dependiendo de los **métodos** y **atributos** que tengan por dentro.

# **Instance Of en JavaScript con instancias y prototipos**

El operador **instanceof** verifica si un objeto en su cadena de prototipos contiene la propiedad prototype de un constructor.