La memoria en JavaScript funciona de la siguiente manera:

* 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

**Cómo es el almacenamiento de objetos en JavaScript**

Cuando creamos variables en JavaScript (aplicable a casi cualquier otro lenguaje), ejecutamos 2 procesos:

1. El primero es la **inicialización**, es decir, le decimos a JS que vamos a crear una nueva variable con un nombre en específico.

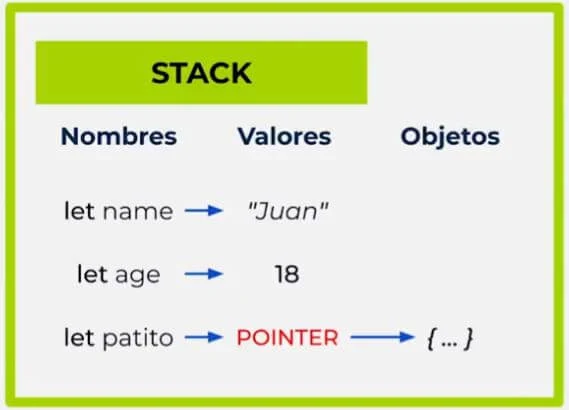
let name;

1. Lo segundo es la asignación: le indicamos a JavaScript que esa variable que generamos con ese nombre en específico tiene un valor.

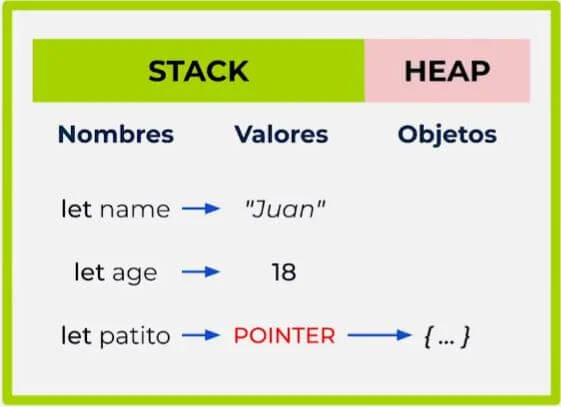
``` let name; // Inicialización name = "pepito"; // Asignación

let age = 28; ```

El nombre de las variables y el valor de estas se almacenan en la memoria **stack,** excepto cuando trabajamos con objetos.



En memoria, el nombre de las variables apuntan a sus respectivos valores, sin embargo, cuando el valor a almacenar es un objeto, apuntan a otro apuntador (*pointer* o puntero) y este es el que en realidad apuntará al objeto en sí el cual se encontrará almacenado en la memoria **heap**.



**Cuál es la forma incorrecta de copiar objetos**

Si intentamos copiar un objeto en otra variable de esta manera:

``` const juanita = { // ORIGINAL age: 15, email: "juanita@juanita.com" }

const nath = juanita; // COPIA ```

Cuando intentemos editar los valores de los atributos del objeto copia, los atributos del objeto original se verán igualmente afectados:

``` // Objeto original antes console.log(juanita); // { age: 15, email: 'juanita@juanita.com' }

// Editamos sólo el objeto copia nath.age = 20; nath.email = "nath@email.com" console.log(nath); // { age: 20, email: 'nath@email.com' }

// Objeto original después de editar el objeto copia: console.log(juanita); // { age: 20, email: 'nath@email.com' } ```

Lo anterior sucede porque cuando copiamos un objeto lo que en realidad estamos copiando es su referencia en la memoria, en otras palabras copiamos a su apuntador o ***pointer***. Por ello, al modificar los valores de las propiedades de la copia de un objeto también afectamos al original.

Entonces, ¿Cómo solucionamos esto? Tenemos 2 formas de hacerlo en JavaScript: el shallow copy y el [deep copy](https://platzi.com/clases/2419-javascript-poo-intermedio/39815-deep-copy-con-recursividad/).

El Shallow Copy (copia superficial) 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.

## Shallow copy con el bucle for

Podemos copiar las propiedades de un objeto en otro haciendo uso del bucle for:

``` const obj1 = { a: "a", b: "b" }

const obj2 = {}

for (propiedad in obj1) { obj2[propiedad] = obj1[propiedad]; } ```

Si deseáramos modificar los valores de los atributos del objeto copia, el objeto original no se ve afectado:

``` obj2.a = "AAA"; obj2.b = "BBB";

console.log(obj2); // { a: 'AAA', b: 'BBB' } console.log(obj1); // { a: 'AAA', b: 'BBB' } ```

Pero, si hay objetos dentro del objeto original (nested objects) el objeto original sí se vería afectado ante las modificaciones hechas en dichos sub objetos:

``` const obj1 = { a: "a", b: "b", c: { d: "d", e: "e" } }

const obj2 = {}

for (propiedad in obj1) { obj2[propiedad] = obj1[propiedad]; }

obj2.a = "atributo a"; obj2.b = "atributo b"; obj2.c.d = "objeto dentro de otro";

console.log(obj2); console.log(obj1);

/> Mensaje en consola { a: 'atributo a', b: 'atributo b', c: { d: 'objeto dentro de otro', e: 'e' } } { a: 'a', b: 'b', c: { d: 'objeto dentro de otro', e: 'e' } }/ ```

## Shallow copy con Object.assign

El Object.assign nos permite realizar el mismo shallow copy que podemos hacer con el bucle for.

``` const obj1 = { a: "a", b: "b", c: { d: "d", e: "e" } }

const obj3 = Object.assign({}, obj1);

// Con esto podemos crear copias exactas console.log(obj1); // { a: 'a', b: 'b', c: { d: 'd', e: 'e' } } console.log(obj3); // { a: 'a', b: 'b', c: { d: 'd', e: 'e' } }

// Sin embargo, si hacemos modificaciones en los nested objects... obj1.c.d = "COPIA DESDE EL OBJ1";

// se verán afectados los demás objetos copiados console.log(obj3); // { a: 'a', b: 'b', c: { d: 'COPIA DESDE EL OBJ1', e: 'e' } } ```

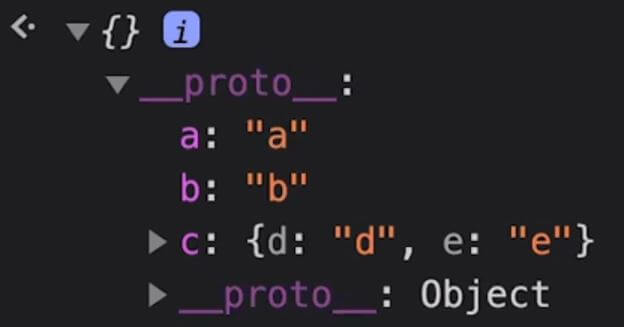
Aun así, tendremos los mismos problemas si el objeto original posee **nested objects**.

## Object.create

Nos permite crear un objeto que tenga como parte de su prototipo los atributos de otro objeto:

``` const obj1 = { a: "a", b: "b", c: { d: "d", e: "e" } }

const obj4 = Object.create(obj1); ```



Hasta ahora hemos podido resolver parcialmente el problema de copiar objetos, ya que aún tenemos inconvenientes cuando los objetos originales tienen anidados otros objetos. Tratemos de resolver esto con [JSON.parse y JSON.stringify](https://platzi.com/clases/2419-javascript-poo-intermedio/39813-jsonparse-y-jsonstringify/). 👨‍💻

Con JSON.stringify podemos convertir un objeto en un string y lo inverso con JSON.parse. Podríamos **usar ambos métodos para lograr copiar un objeto en otro**.

``` const obj1 = { a: 'a', b: 'b', c: { d: 'd', e: 'e', } }

// Lo convertimos en String y el resultado lo asignamos en un variable: const stringifiedComplexObj = JSON.stringify(obj1); // Lo convertimos a objeto y lo asignamos al nuevo objeto: const obj2 = JSON.parse(stringifiedComplexObj);

console.log(obj1); console.log(obj2);

// Si hacemos modificaciones en un de los objetos... obj2.c.d = "nested object"; obj2.c.e = "nested object"; // El objeto original no se vería afectado console.log(obj1); console.log(obj2); ```

De esta forma, aunque el objeto original tenga objetos anidados, es posible crear un nuevo objeto con las mismas propiedades del original sin que este se vea afectado ante modificaciones en los objetos copias.

**Problemas para copiar un objeto con métodos**

Con lo anterior pareciera que finalmente habíamos logrado dar solución a los convenientes que nos daba copiar objetos en otros. Sin embargo, tanto JSON.parse como JSON.stringify no saben trabajar con métodos, lo cual sería un nuevo inconveniente:

``` const obj1 = { a: 'a', b: 'b', c: { d: 'd', e: 'e', }, editA() { this.a = 'Abcd' } };

const stringifiedComplexObj = JSON.stringify(obj1);

console.log(stringifiedComplexObj); // "{\"a\":\"a\",\"b\":\"b\",\"c\":{\"d\":\"d\",\"e\":\"e\"},\"f\":[1,\"2\",3]}"

const obj2 = JSON.parse(stringifiedComplexObj);

console.log(obj2); // {a: "a", b: "b", c: {d: "d", e: "e"}} ```

Para ello necesitamos utilizar Deep Copy para poder obtener una correcta copia de objetos. Antes de entrar en este concepto, debemos conocer [qué es recursividad](https://platzi.com/clases/2419-javascript-poo-intermedio/39814-que-es-recursividad/). ¡Vamos a ello! 🤔👩‍💻

**Qué más necesitas saber acerca de 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.

**Descripción**

* Los objetos Boolean, Number y 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 replacer.
* 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) solo tendrán serializadas sus propiedades enumerables.

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

**Qué más necesitas saber acerca de 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(), asi que GG mi rey
* 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.

## Recursividad

La recursividad es cuando una función se llama a sí misma y esta genera una nueva ejecución de la función. Esto sucede reiteradamente hasta que cumpla o no con cierta validación que nosotros declaremos para que deje de llamarse a sí misma en algún punto.

Normalmente, pensaríamos que este tipo de problemas lo podemos resolver con condicionales y/o bucles. Veamos el siguiente ejemplo:

``` // PROBLEMA: // Deseamos imprimir una serie de números desde el 0 hasta n números. En este caso // hasta el 4 let numerito = 0; // Declaramos desde el número que deseamos partir.

// Usamos un bucle while para repetir este proceso hasta que se cumpla la condición: while(numerito < 5) { console.log(numerito); numerito++; } ```

Así resolveríamos este problema con recursividad:

// Función recursiva: function recursiva(numerito) { // Recive un número console.log(numerito); // Imprimimos en consola el número if (numerito < 5) { // Evalua si es menor a 5 // Llamamos nuevamente a nuestra función enviandole el número siguiente: return recursiva(numerito + 1); } else { // La función deja de llamarse a sí misma: return 5; } }

**¿Por qué escribir programas recursivos?**

Son más cercanos a la descripción matemática.

Generalmente más fáciles de analizar

Se adaptan mejor a las estructuras de datos recursivas.

Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

**¿Cuándo SÍ es factible de utilizar recursividad?**

Para simplificar el código.

Cuando la estructura de datos es recursiva. Ejemplo: árboles.

**¿Cuándo NO es factible utilizar recursividad?**

Cuando los métodos usen arreglos largos.

Cuando el método cambia de manera impredecible de campos.

Cuando las iteraciones sean la mejor opción

Con el Deep Copy podemos **generar copias de objetos sin importar que estos posean objetos anidados o métodos dentro**.

**Aplicando Deep Copy en JavaScript**

Veamos el siguiente ejemplo:

``` // OBJETO ORIGINAL const studentBase = { name: undefined, email: undefined, age: undefined, approvedCourses: undefined, learningPaths: undefined, socialMedia: { twitter: undefined, instagram: undefined, facebook: undefined, }, hello() { console.log("Hello, World!"); } };

// FUNCIÓN RECURSIVA function isObject(subject) { // Comprueba si es un objeto return typeof subject == "object"; // Devuelve true o false }

function isArray(subject) { // Comprueba si es una Array return Array.isArray(subject); // Devuelve true o false }

// FUNCIÓN RECURSIVA // Recibe un parametro que puede ser un objeto, array u otro tipo de dato function deepCopy(subject) { let copySubject; // Esta variable se convertira en array, objeto u otro tipo de dato

const subjectIsObject = isObject(subject); // ¿El parámetro es objeto? const subjectIsArray = isArray(subject); // ¿El parámetro es array?

if (subjectIsArray) { // Si es array... copySubject = []; // Asignamos un array vacío donde iremos copiando 1 a 1 los datos } else if (subjectIsObject) { // Si es un objeto... copySubject = {}; // Asignamosun objeto vacío donde iremos copiando 1 a 1 los atributos } else { // Sino es array u objeto... // Entonces es un tipo de dato que se puede copiar sin problemas, retornamos dicho // dicho dato y terminamos con la ejecución de la fucnción. return subject; }

// Continuamos con la ejecución de la función si el parámetro fue array u objeto:

for (key in subject) { // Recorremos cada uno de los atributos o datos del objeto o array // Comprueba si hay un objeto dentro del índice o atributo: const keyIsObject = isObject(subject[key]);

if (keyIsObject) { // Si es verdad que hay un objeto dentro...

// Invocamos recursivamente la misma función:

copySubject[key] = deepCopy(subject[key]); // 👀🔄

} else { // Sino...

if (subjectIsArray) { // Si el parámetro recibido por la función deepCopy es Array...

// Agregamos el elemento a la variable a retornar al final de la función:

copySubject.push(subject[key]);

} else {

// sino, significa que es objeto el parámetro y además no hay objetos anidados

// en el elemento actual dentro del recorrido del bucle for, por tanto, asignamos

// dicho elemento como valor a la propiedad correspondiente:

copySubject[key] = subject[key];

}

}

}

return copySubject; // Finalmente retornamos el objeto/array copia } ```

Generemos un objeto copia utilizando la función recursiva e intentemos realizar modificaciones en el objeto copia y original:

``` // OBJETO COPIA const juan = deepCopy(studentBase);

// MODIFICACIONES EN EL OBJETO ORIGINAL studentBase.socialMedia.twitter = "@student\_twitter"

// MODIFICACIONES EN EL OBJETO COPIA juan.socialMedia.instagram = "@juanDC"

// VEAMOS EN CONSOLA LAS DIFERENCIAS DEL OBJETO ORIGINAL Y LA COPIA console.log(studentBase); console.log(juan);

/*> Mensaje en consola { name: undefined, email: undefined, age: undefined, approvedCourses: undefined, learningPaths: undefined, socialMedia: { twitter: '@student\_twitter', 👈👈 👀 instagram: undefined, facebook: undefined }, hello: [Function: hello] 👈👈 FUNCIÓN } { name: undefined, email: undefined, age: undefined, approvedCourses: undefined, learningPaths: undefined, socialMedia: { twitter: undefined, instagram: '@juanDC', 👈👈 👀 facebook: undefined }, hello: [Function: hello] 👈👈 FUNCIÓN }*/ ```

Podemos notar que los cambios en un objeto no afecta en los valores de las propiedades del otro. Logramos crear una copia de un objeto que no esté condicionada a que si el objeto original tiene objetos anidados o si tiene métodos dentro.

Conozcamos ahora cómo emplear la [abstracción en JavaScript con simplemente objetos](https://platzi.com/clases/2419-javascript-poo-intermedio/40092-abstraccion-con-objetos-literales-y-deep-copy/), es decir, sin utilizar clases. 🤔🚀