

# BAZ - Javascript Junior

## Funciones

### ¿Qué es una función?

una función en JavaScript es un bloque de código que diseñamos con el fin de ejecutar una tarea determinada, este bloque de código lo almacenamos dentro de un comando o palabra clave que podemos nombrar como queramos y que para ejecutarse debe ser invocada o llamada en cualquier parte de nuestro código.

### ¿Cómo se declara una función?

Para escribir una función debemos usar la palabra clave `function`, seguimos con el nombre que queremos darle, continuamos con paréntesis donde se podrán incluir parámetros que sean usados por el bloque de código y terminamos con llaves, dentro de las llaves escribiremos el bloque de código a ejecutar, así:

```
function operacion(parametro1,parametro2) {  
  // código a ejecutar  
}
```

### ¿Para qué usamos las funciones?

La utilidad de las funciones en JavaScript es la de reutilizar bloques de código, pues solo deberemos definir nuestra función una vez e invocarla las veces que sean necesarias, así mismo, mediante los parámetros añadiremos dinamismo a nuestro código, pero, además debemos tener en cuenta que los parámetros serán

manejados dentro de la función como variables locales lo que quiere decir que su contenido no es accesible fuera de la función, de igual manera las variables definidas dentro de una función sólo serán accesibles dentro de esa función, volvamos al ejemplo anterior:

```
// el código NO accederá a la variable suma de la función  
operacion
```

```
function operacion(parametro1,parametro2){  
  Let suma = parametro1 + parametro2;  
  // el código SI accederá a la variable suma  
  
}
```

```
// el código NO accederá a la variable suma de la función  
operacion
```

Lo anterior quiere decir que podemos declarar una variable suma fuera de la función operación y que sus valores sean siempre independientes, a esta cualidad de las funciones en JavaScript, le llamamos local scope, que por ahora definiremos como el campo de acceso a las variables.

Ahora, ya sabemos declarar una función y entendemos el local scope propio de esta herramienta, pero, ¿como se ejecuta una función?, para llamar una función usaremos el nombre de la función seguido de paréntesis, dentro de los paréntesis si es el caso, indicaremos los argumentos que evaluara la función separados por comas, continuando con el ejemplo, para llamar la función operación lo haremos de la siguiente manera.

```
operacion(2,4);
```

Nuestra función tomará los valores 2 y 4, creará localmente la variable suma a la que le asignara el valor de dicha adición, si agregamos dentro de nuestra función la siguiente línea `console.log(suma);` obtendremos el resultado esperado.

---

## Return

Cuando las funciones alcanzan la palabra clave return detienen su ejecución, por otro lado, si la llamada a la función se realizó desde un bloque de código, el código continuará justo después de la llamada realizada a la función respectiva al alcanzar el return.

Podemos añadir a la palabra return la operación, valor o variable obtenido de nuestra función con el fin de utilizarlo como una nueva variable, ejemplo:

```
function operacion(parametro1,parametro2){  
    Let suma = parametro1 + parametro2;  
    Return suma;  
}
```

```
Let resultado = operacion(2,4);
```

Gracias al return, devolvemos el valor suma y lo asignamos a la variable resultado.

---

## Actividad

En esta actividad, en Visual Studio Code, crea una función llamada operaciondivision que reciba dos parámetros, nombrados parametro1 y parametro2, en el bloque de código de la función crea una variable llamada div asignándole el valor de la operación parametro2 / parametro1.

Ahora añade la palabra return div, para finalizar, por fuera de la función declara una variable llamada resultado y asigna como resultado a la llamada de la función operaciondivision(4/2), imprime en consola la variable resultado.

```
function operaciondivision(parametro1,parametro2){  
    let div = parametro2 / parametro1;  
    return div;  
}  
const resultado = operaciondivision(4,2);  
console.log(resultado);
```

## Arrow function

Ahora que ya conocemos la estructura de las funciones, podemos continuar con las funciones flecha o Arrow Functions ¿en qué consisten? En pocas palabras las funciones flecha son una alternativa compacta para escribir una función.

El siguiente ejemplo define una función tradicional que evalúa la suma de dos argumentos y lo devuelve.

```
function suma(x, y) {  
    return x + y;  
}  
  
console.log(suma(3, 4));
```

Siendo equivalente, ahora usaremos la función flecha, para notar sus diferencias.

```
let suma = (x, y) => x + y;  
  
console.log(suma(3,4));
```

En esta función, sólo se realizará una operación, así que no es necesario añadir la palabra clave return, sin embargo, en caso de necesitar realizar más de una operación podrás añadir un bloque de código dentro de llaves, aquí sí deberás escribir la palabra clave return.

Pero la diferencia más notable entre una función tradicional y

una función flecha es que en la función flecha no deberemos escribir la palabra clave function usaremos en cambio la flecha (=>) entre los parámetros y la expresión, además, dependiendo del caso podremos prescindir de los paréntesis, llaves y la palabra clave return, pero así mismo estos mismos elementos podrán ser obligatorios ¿como podemos determinar su obligatoriedad?

1. Si tenemos varios parámetros o ningún parámetro deberás usar los paréntesis.

```
// Función tradicional
function sum(a, b){
  return a + b + 100;
}

// Función flecha
const sum = (a, b) => a + b + 100;

// Función tradicional (sin argumentos)
let a = 4;
let b = 2;
function sum(){
  return a + b + 100;
}

// Función flecha (sin argumentos)
let a = 4;
let b = 2;
const sum = () => a + b + 100;
```

2. Si el bloque de código de nuestra función flecha necesita varias líneas de código, deberemos usar las llaves y obligatoriamente, la palabra clave return, pues debemos definir a la función qué o cuándo devolver un dato o variable.

```
// Función tradicional
function sum(a, b){
  let chuck = 42;
  return a + b + chuck;
}
```

```
// Función flecha
const sum = (a, b) => {
  let chuck = 42;
  return a + b + chuck;
}
```

Para efectos prácticos describiremos la sintaxis básica de las funciones flecha.

Un parámetro. Con una expresión simple no se necesita return:

**Parámetro => expresión**

Varios parámetros requieren paréntesis. Con una expresión simple no se necesita return:

**(parámetro1, parámetro2) => expresión**

Las declaraciones de varias líneas requieren llaves y return:

```
Parámetro => {
  let a = 1;
  return a + b;
}
```

Varios parámetros requieren paréntesis. Las declaraciones de varias líneas requieren llaves y return:

```
(parámetro1, parámetro2) => {
  let a = 1;
  return a + b;
}
```

---

## Early Return

Ya que sabemos que la palabra clave `return`, permite detener la ejecución de una función y devolver un valor o variable para ser utilizado en cualquier parte de nuestro código, el `early return` es una nueva técnica donde lograremos reducir el número de condicionales usados dentro de una función, añadiendo mejoras y simplificando la lectura de nuestro código.

El `early return` es la manera de escribir funciones donde el resultado positivo de la condición se devuelve al final de la función y el resto de código termina con la ejecución al implementar la palabra clave `return`. es decir si se cumple la condición usamos la palabra clave `return` y de esta manera no se ejecuta el resto del código:

```
const someOperations = () => {  
  let param1 = 2;  
  let param2 = 2;  
  if(param1 === param2 ) return param1;  
  param1 = 5;  
  param2 = 7;  
  return param1;  
}  
  
console.log(someOperations()); // 2
```

como se puede observar en el ejemplo anterior dado que se cumple la condición `if` se aplica el `fast return` y no se realiza la reasignación de valores de `param1` y `param 2`. de esta manera podemos tener varios `early return` en una función, donde la gran ganancia de esta técnica es que evitamos que se ejecute código que no necesitamos únicamente las que cumple las condiciones, el siguiente podría ser un ejemplo de una función que retorna si `true` o `false` si un número es par como se presenta:

```
const esPar = (parametro1) => {  
  if(parametro1 % 2 === 0) return true;  
  if(isNaN(parametro1 % 2)) return 'el parámetro dado no
```

```
    es un número';  
    return false  
  }  
  console.log(esPar(2)) // true  
  console.log(esPar('hola')) // el parámetro dado no es un  
  número  
  console.log(esPar(3)) // false
```

La función isNaN es un método nativo expuesto por javascript para determinar si un valor es o no un número.

---

## Actividad

En visual Studio Code, declara una variable comparación y asigna a esta variable una función flecha con dos parámetros, el bloque de código , deberá evaluar mediante condicionales si el valor de los parámetros es igual o diferente, en cuyo caso deberás devolver el resultado correspondiente.

Ahora escribe console.log(comparación(2,10));, ¿qué resultado obtuviste?, realiza más pruebas e intenta comparar palabras.

```
const comparación = (parámetro1, parámetro2) => {  
  if(parámetro1 === parámetro2){  
    return true  
  } else {  
    return false  
  }  
}  
console.log(comparación(2,10)) // false
```

## Iteradores

La pregunta que debemos responder para adentrarnos en este tema es **¿que se debe hacer si necesitamos que un determinado bloque de código se repita una cierta cantidad de veces?**, para dar solución JavaScript integra los bucles, donde mientras se cumpla



cierto criterio, se repetirá el código determinado ¿cuales son estos bucles?

Primero debemos entender los fundamentos, los bucles en JavaScript se refieren a hacer una cosa varias veces, denominado como iteración en el mundo del desarrollo, ¿cuáles son las características de un bucle?

**Son tres las características de un bucle, un contador que inicia en un determinado valor, este será el punto de partida para el bucle, el segundo es la condición de salida, que no se refiere a otra cosa que el criterio bajo el cual terminará el bucle y por último un iterador que modificara el valor del contador hasta que alcanza la condición de salida.**

## Bucle For

El primero de los bucles que abordaremos será el bucle for, para declararlo, utilizaremos la palabra clave for seguida de paréntesis, dentro de estos paréntesis determinaremos un número que actuará como el contador y punto de partida que aumenta cada vez que se repita el bucle, también dentro de los paréntesis estableceremos una condición de salida, un iterador que podría aumentar o disminuir al contador en cada repetición del bucle y por último llaves, donde escribiremos el código que se ejecutará en cada repetición, así:

```
For (inicializador; condición de salida; iterador){  
  // código a ejecutar  
}
```

---

## Bucle while y do while

Pero como ya sabemos, for no es el único tipo de bucle en JavaScript, existen además, los bucles while y do while ¿en que se diferencian al bucle for?, en los bucles while y do while debemos declarar el contador antes del bucle y el iterador se declara al final de todo el bloque de código, que se ejecuta siempre y cuando se cumpla la condición que en este caso se

escribe anteponiendo la palabra clave while.

persisten las tres características de los bucles, ahora veamos la estructura del bucle while y seguiremos con la diferencia con el bucle do while.

#### **Inicializador**

```
While (condición){  
    // código a ejecutar  
  
    iterador  
}
```

Por otro lado, el bucle do while es bastante similar pero modifica la estructura anterior, debemos declarar el contador antes del bucle, iniciamos el bucle con la palabra clave do, y entre las llaves declaramos el bloque de código y el iterador, Después de las llaves al final se declara la palabra clave while y la condición a evaluar lo que afecta la forma de ejecutar el código, pues en el bucle for y while primero se evalúa la condición, en este caso, primero se ejecuta el código y después se evalúa si cumple o no la condición, aquí su estructura.

#### **Inicializador**

```
Do{  
    // código a ejecutar  
  
    iterador  
} While (condición)
```

## **Bucle Infinito**

Es muy importante tener claro qué en cualquier bucle debemos estar seguros de que la condición que declaremos se cumpla, de lo contrario habremos creado un bucle infinito que bloqueara el código e impedirá que puedas seguir ejecutando el flujo de código.

---

## Actividad

Para aplicar lo aprendido, deberás tomar el bucle while y modificarlo para crear uno do while, ejecútalo en Visual Studio Code compara los resultados, por último, elimina el iterador y vuelve a ejecutar el código.

```
Let i=0
while(i<10){
  Texto += "el número es " + i;
  console.log(texto);
  i++
}
```


## Arreglos (Crud, matrices, etc)

En un array puede existir una mezcla de los diferentes tipos de datos, claro, guardando las características de declaración que corresponda a cada tipo.

Así mismo, las listas o arrays son modificables, podremos añadir, actualizar o borrar cualquier dato contenido, pero primero, ¿cómo podemos acceder a un dato en una lista?

Para esto debemos comprender que las listas se miden en índices numéricos, así mismo, que el índice de inicio es 0, lo que quiere decir, que el primer dato de nuestro array será el índice 0, el segundo dato será el índice 1 y así consecutivamente dependiendo del largo de nuestro array, aquí un ejemplo:

```
a = [ "a", "b", "c", "d", "e" ]
```



The diagram illustrates the indexing of the array 'a'. Below the array elements, indices 0 through 5 are listed. Orange arrows point from each index to its corresponding element in the array: index 0 points to 'a', index 1 to 'b', index 2 to 'c', index 3 to 'd', index 4 to 'e', and index 5 points to the end of the array.

En ese sentido, para acceder a un elemento dentro de un array deberemos especificar el índice numérico con la siguiente estructura, el nombre de la variable que contiene la lista y entre corchetes el índice del valor que nos interesa:

```
console.log(a[5]);  
// imprimirá la letra e de la lista a.
```

Comprendido lo anterior, en JavaScript el tipo de dato lista, posee propiedades y métodos propios muy útiles para nuestro código. Analizaremos detalladamente la propiedad y sus métodos más utilizados.

---

## Propiedad length

El uso de esta propiedad en cualquier array nos devolverá el número de elementos contenidos.

En nuestro ejemplo del array `a`, si utilizamos la propiedad `.length`, obtendremos el número 5.

```
console.log(a.length);  
// imprimirá el número 5
```

Esta propiedad es mayormente utilizada como herramienta de evaluación.

---

## Método forEach

En este caso, el método `forEach` permite aplicar una función para cada elemento del array en orden ascendente desde el índice 0, puede escribirse en lugar de un bucle `for`.

La función llamada dentro del método `forEach` se invoca con tres argumentos.

1. El valor del elemento
2. El index del elemento
3. El array que está siendo recorrido.

Es relevante entender que los argumentos que deseamos sean evaluados y manejados por el método `forEach` debe ser declarados en la función, cuyo orden será:

```
//función flecha  
forEach((elemento, index, array) => expresión  
  
//función tradicional  
forEach(function(element, index, array){expresion})
```

Si bien puede reemplazar un bucle `for`, el método `forEach` no puede ser detenido prematuramente, en caso de que necesites implementar un `early return`, deberás aplicar un bucle tradicional u otros métodos específicos.

Como ejemplo creamos un array llamado `listal` que contenga dos nombres y dos números aleatorios.

```
let listal = ["adam", "valeria", 5 , 3];
```

Ahora, a `listal` apliquemos el método `forEach` y dentro de los paréntesis definimos una función flecha que tome el valor del elemento y el `index` como parámetros e imprima en consola el valor del `index` y el elemento en su lugar separados por un espacio.

```
const listal = ['cristian', 'yina', 'andrea'];  
  
listal.forEach((elemento, index) => {  
    console.log(index + ' ' + elemento)}  
);
```

```
// 0 cristian
// 1 yina
// 2 andrea
```

---

## Método find

Este método evalúa una condición en orden ascendente desde el índice 0, en dicho recorrido devolverá el primer elemento que cumpla con esa condición. Si ningún elemento cumple la condición devolverá undefined.

El método find comparte características del método forEach pues la evaluación se realizará por medio de una función, ya sea flecha o tradicional, que podrá incluir los mismos argumentos en el mismo orden:

1. El valor del elemento.
2. El index del elemento.
3. El array que está siendo recorrido.

Continuando el ejemplo con la `lista1`, aplicaremos el método find declarando una función que reciba el valor del elemento como parámetro y evalúe si ese valor es mayor al número 1.

```
console.log(lista1.find(elemento => elemento >1));
// imprimirá el número 5
```

Cómo mencionamos este método devuelve el primer valor que cumpla la condición en ese sentido, una vez evalúe el número 5 del array terminará el recorrido devolviendo el valor correspondiente.

---

## Método filter

En este método igual que en los anteriores se evalúa una condición en orden ascendente desde el índice 0, recorrerá la totalidad del array y devolverá un nuevo array con todos los valores que cumplan dicha condición.

Continuando con el ejemplo, crearemos una nueva variable que nombraremos lista2 y le asignaremos la aplicación del método filter a la lista 1 que reciba como parámetro el elemento y evalúe si el valor es mayor al número 1.

```
let lista2 = lista1.filter(elemento => elemento > 1);  
// lista2 será una nueva lista conformada por los valores 5  
y tres.
```

Cómo es notable se evaluará la condición mediante funciones flecha y tradicionales que cumplan las características de los métodos anteriores.

---

## Método map

En este caso, el método map recorrerá la totalidad del array, aplicará a cada elemento una función y con el resultado creará un nuevo array, la función que determinemos, como es entendible, debe cumplir con las características y argumentos determinados en los anteriores métodos.

Crearemos ahora, un nuevo array que nombraremos listaNumeros y le asignaremos dentro de los corchetes los números 5, 6, 7 y 8.

```
let listaNumeros = [5, 6, 7 , 8];
```

Crearemos una variable llamada listaDoble, le asignaremos la aplicación del método map a la listaNumeros que reciba como

parámetro el elemento y realice la multiplicación del elemento por el número 2.

```
let listaDoble = listaNumeros.map(elemento => elemento *  
2);  
// listaDoble corresponderá a [10, 12, 14 , 16];  
// listaNumeros continuará con su valor original
```

---

## Método sort

En este caso, el método sort retorna un array ordenado, donde si no se inyecta una función a este método se convertirá cada posición del string a un array y dando el orden dependiendo de su posición unicode, en el caso que se inyecte una función al método .sort lo elementos del array serán ordenados dependiendo del valor de retorno de la función siendo val1 y val2 los elementos comparados dentro de la función de comparación:

- Si comparando val1 y val2 es menor que 0, val1 tomará un índice menor que val2. Es decir, val1 viene primero que val2.
- Si comparando val1 y val2 es mayor que 0, val1 tomará un índice mayor que val2. Es decir, val1 viene después de val2.
- Si comparando val1 y val2 es igual que 0 no se intercambian posiciones.

nota: la función inyectada el método sort siempre debe retornar un valor numérico positivo, negativo o cero y de esta manera sort realiza la organización del array.

Crearemos ahora, un nuevo array que nombraremos listaNumeros y le asignaremos dentro de los corchetes los números 5, 6, 7 y 8.

```
const listaNumeros = [8,0,3,1,9];
```

Crearemos una variable llamada listaOrdenada, le asignaremos la



aplicación del método `sort` a la `listaOrdenada` que reciba como parámetro el elemento y compare cuál número es mayor.

```
let listaOrdenada = listaNumeros.sort((a,b) => {
  if (a>b){
    return 1
  }
  return -1
});
console.log(listaOrdenada) // [0, 1, 3, 8, 9]
```

---

## Método slice

Este método `slice` retorna un array recortado de una array inicial, este método acepta uno o dos parámetros, si solo se da un parámetro el array de entrada va desde la posición dada hasta el final del array por ejemplo:

```
const list1 = [28,45,12,32,98,72];
const listSlice = list1.slice(2);
console.log(listSlice) // [12, 32, 98, 72];
```

Es decir se recorta desde la posición 2 hasta el final del array.

Cuando se inyecta dos valores al método retorna un array recortado donde el primer parámetro será el índice inicial de recorte hasta el segundo parámetro, por ejemplo:

```
const list1 = [28,45,12,32,98,72];
const listSlice = list1.slice(1,4);
console.log(listSlice) // [45, 12, 32];
```

## Actividad

Declararemos una variable `list1` a la cual le asignaremos la lista de números 28, 45 y 19, ahora declaremos otra variable `list2` a la cual le asignaremos una lista de nombres, Juan, Carlos, Cristian, lo primero que haremos será comparar si las dos listas tienen la misma longitud e imprimiremos ese resultado en consola.

```
Const list1 = [28,45,19];
Const list2 = ["Juan", "Carlos", "Cristian"];

if (list1.length === list2.length) {
  console.log(true);
} else{
  console.log(false);
}

//el resultado en nuestro ejemplo será true, siempre y
cuando conserven la misma cantidad de elementos en cada
lista.
```

Prueba agregando elementos a las listas para obtener otro resultado.

Ahora, necesitamos tomar cada elemento de `list1` e imprimir ese elemento en consola usaremos `foreach`.

```
list1.forEach(element => {
  console.log(element)
});
// el resultado será en consola en orden, 28, 45 y 19
```

prueba agregar otro tipo de lógica como algún tipo de condicional, que verifique si cada elemento es mayor o menor a otro.

Otro ejercicio será, encontrar en `list2` el primer elemento que su longitud sea mayor a 7, usaremos dentro del método `find`, la

propiedad length para comparar si es mayor o igual a 7

```
list2.find(element => {  
    if (element.length >= 7) {  
        console.log(element)  
    }  
});  
// con el método find realizaremos un recorrido por toda la  
lista2, a cada elemento se evaluará si su longitud es mayor  
o igual a 7, en nuestro ejemplo será Cristian quién cumpla  
nuestra condición.
```

Ahora declararemos una variable nuevaLista donde guardaremos el resultado de filtrar los elementos de list2 que su inicial sea C e imprimiremos en consola nuevaLista

```
var nuevaLista = list2.filter(element => element[0] == "C")  
console.log(nuevaLista);  
  
// el resultado sera una nueva lista con los nombres Carlos  
y Cristian
```

Prueba cambiando la condición a "J" Para ver la nueva lista.

## Objetos literales (Crud, Mezclando arreglos y objetos, etc)

Ya que entendemos más ampliamente el concepto del tipo de dato arrays y sus métodos, es el turno de profundizar sobre el tipo de dato objeto.

En este momento entendemos que a una variable podemos asignarle un valor, a manera de ejemplo, El tipo de dato objeto o object permite almacenar múltiples variables y su respectivo valor, el nombre correcto de esas propiedades son clave y valor y la mejor forma de representarlo es con objetos reales como un vehículo. Dicho vehículo, tiene entre otras características, un modelo, una marca y un color, en JavaScript podemos crear un tipo de dato object llamado vehículo, crear las tres claves mencionadas

y asignarles un valor correspondiente, así:

```
const vehículo = {  
  modelo: 2022,  
  marca: "chevrolet",  
  color: "rojo"  
};
```

Cada conjunto de clave valor se declara con dos puntos y se separa de otro conjunto mediante coma, así mismo como aprendimos con el método prototype, existe la posibilidad de definir funciones dentro de un objeto.

Por otro lado, igual de relevante al acceso de un dato específico en un array, es aprender a acceder a un dato o propiedad específica de nuestro Object, para lo que cada propiedad de nuestro object adoptará las característica de un método, en ese sentido, para imprimir en consola el color de nuestro vehículo usaremos el método color, así:

```
console.log(vehiculo.color);  
// imprimirá rojo.
```

Como es de esperarse al igual que él array, el tipo de dato object tiene métodos propios bastante útiles que facilitan el manejo y aplicación de este tipo de dato en nuestro código.

---

## Método entries

Este método devuelve un array, que contendrá a su vez otro array por cada uno de los pares clave: valor obtenidos del objeto. Por ejemplo, para aplicarlo a nuestro objeto vehículo, será así:

```
let objetolista = Object.entries(vehículo);  
console.log(objetolista) // [Array(2), Array(2), Array(2)]  
console.log(objetolista[0]) // ['modelo', 2022]
```

```
console.log(objetolista[1]) // ['marca', 'chevrolet']  
console.log(objetolista[2]) // ['color', 'rojo']
```

Debo aclarar que para aplicar métodos propios del tipo de dato Object, la palabra clave Object incluye la letra "o" mayúscula.

---

## Método keys

Por otro lado, el método keys, devolverá un array que contendrá todas las claves obtenidas del objeto, será aplicable igual que el método entries, así:

```
let objetoKeys = Object.keys(vehiculo);  
//objetoKeys = [modelo, marca, color]
```

---

## Método values

Como el método anterior, values devolverá un array, pero esta vez con los valores obtenidos del objeto.

```
let objetoValues = Object.values(vehiculo);  
// objetoValues = [2022, chevrolet, rojo]
```

---

## Método create

Podríamos llegar a decir que el presente método, está ligado al método prototype, pues, con el método create, definiremos un nuevo objeto usando otro objeto como prototipo, Por ejemplo:

```
const camion = Object.create(vehículo);
```

al usar el objeto vehículo como prototipo, heredará las propiedades modelo, marca, color y podrá asignarles a cada una de ellas el valor que corresponda a su caso, sin embargo, también podrá crear nuevas propiedades, siguiendo la siguiente estructura.

```
//Aquí se determinarán las características heredadas del  
objeto prototipo vehículo.
```

```
camion.modelo = 2015;  
camion.marca = "renault";  
camion.color = "blanco";
```

```
//Aquí se creará una propiedad del objeto camion
```

```
camion.capacidadCarga = "3 toneladas";
```

Ahora continuando la idea, el objeto camión podría ser usado como objeto prototipo para otro tipo de vehículo relacionado tal como se expuso en el método prototype.

## Actividad

Con lo aprendido manipula el objeto estudiante:

```
const estudiante = {  
  nombre: "jhon",  
  edad: 29,  
  pais: "colombia"  
};
```

Crea tres variables estudianteEntries, estudianteKeys y estudianteValues y le asignaras a cada uno el metodo correspondiente, para obtener nuevas listas.

```
var estudianteEntries = Object.entries(estudiante);

console.log(estudianteEntries); //[Array(2), Array(2),
Array(2)]
console.log(estudianteEntries[0]); // ['nombre', 'jhon']
console.log(estudianteEntries[1]); // ['edad', 29]
console.log(estudianteEntries[2]); // ['pais', 'colombia']

var estudianteKeys = Object.keys(estudiante);
console.log(estudianteKeys); // ['nombre', 'edad', 'pais']

var estudianteValues = Object.values(estudiante);
console.log(estudianteValues); // ['jhon', 29, 'colombia']

// ahora puedes imprimir de forma independiente cada
variable para ver las particularidades de cada metodo
```