Clasess

Muchas veces vamos a necesitar el crear multiples objetos parecidos, por ejemplo, digamos que estamos haciendo una suerte de base de datos con las personas que conocemos, sus edades, sus gustos musicales, mascotas, etc. Podemos hacerlo de la manera literal, no es dificil, creamos cada uno de los objetos asignandole un nombre a cada uno y listo, el problema con esto es que, no solo es tedioso, sino que también es repetitivo y cuesta tiempo, recuerden que, la programación, intenta hacer las cosas más simples todo el tiempo, y ustedes como programadores, tienen que tener eso siempre como prioridad, la simpleza, y la eficiencia de su codigo.
Para esto, javascript nos proporciona una herramienta que se llama Classes, una clase es un constructor de objetos, que no solo nos permite asignarle propiedades a los objetos en cuestion, sino que tambien podemos asignar funciones que se compartiran con todos los objetos que creemos con una misma class, de esta manera, si por ejemplo, creamos una clase llamada "Persona" todos los objetos que creemos con su constructor, compartiran las funciones que definamos dentro de la misma, por ejemplo:

```
class Personas {
  constructor(nombre,apellido,edad,ocupacion){
   this.name = nombre;
   this.lastName = apellido;
   this.age = edad;
   this.occupation = ocupacion;
  }

  setNombre(nuevoNombre){
     this.name = nuevoNombre;
  }
 }
```

Closures

A diferencia de otros conceptos como funciones, variables u otros, los closures no siempre son utilizados a conciencia y de forma directa, lo mas probable es que los hayan usado muchas veces sin darse cuenta, aprender sobre closures, es mas sobre identificar cuando lo estas utilizando, que aprender un nuevo concepto en si.
Una manera de definirlas sería que son funciones que encapsulan una serie de variables y definiciones locales, que solo son accesibles si son devueltas con el operador return, esto nos permite en cierta manera, tener variables "casi" privadas, no son totalmente así, pero al menos nos asegura el dificil acceso a las mismas

```
const contador = (function(){
  let _count = 0

  function incrementar(){
  return _count++;
  }
  function decrementar(){
  return _count--;
  }
  function valor(){
  return _count;
  }
```

```
  return {
  incrementar,
  decrementar,
  valor
  }
})();
```

Es muy parecido a lo que son classes, y nos permiten tener una suerte de privacidad en las variables que manejamos dentro de las mismas

Callbacks

Un callback es una funcion que es pasada como argumento en otra funcion y es llamada desde la misma, el concepto puede parecer un poco confuso, por eso es mejor ir viendolo con ejemplos practicos, vamos a ver un mismo codigo, expresado de dos maneras distintas, una normal, y la otra con callbacks, así vemos el porque es importante este concepto, y como puede mejorar nuestro codigo.

Vamos a hacer una funcion que haga calculos matematicos, suma, resta, multiplicacion y division:

```
function calculadora(numero1, numero2, operacion){
  if(operacion==="suma"){
    return numero1 + numero2;
  }else if(operacion==="resta"){
    return numero1 - numero2;
  }else if(operacion==="multiplicacion"){
    return numero1 * numero2;
  }else if(operacion==="division"){
    return numero1 / numero2;
  }
}
```

Ahora vamos a escribir las mismas funciones, pero con callbacks:

```
let suma = function(numero1, numero2){
  return numero1 + numero2;
}
let resta = function(numero1, numero2){
  return numero1 - numero2;
}
let multiplicacion = function(numero1, numero2){
  return numero1 * numero2;
}
let division = function(numero1, numero2){
  return numero1 / numero2;
}

let calculadora = function(numero1, numero2, operacion){
  return operacion(numero1, numero2);
}
```

Nosotros estamos pidiendo en la funcion calculadora, tres argumentos, pero que pasa si alguien en

vez de poner una funcion, pone una string? Va a dar error, entonces, para este caso, como en todos los casos en los que utilizamos callbacks, tenemos que asegurarnos que lo que ingresen como argumento en "operacion", sea una funcion, esto lo hacemos de la siguiente manera:

```
let calculadora = function(numero1, numero2, operacion){
  if(typeof operacion === "function"){
  return operacion(numero1, numero2);
  } else {
    return "la operacion ingresada no es valida";
  }
}
```

Recursividad

Una función recursiva es una función que en alguna parte de su codigo se llama así misma, es un concepto facil de entender, pero no tanto de aplicar, por ejemplo, una funcion recursiva pura podría ser la siguiente:

```
function saludar() {
  console.log("Hola!");
  return saludar();
}
```

Esto nos daría un bucle infinito, que acabaría solamente cuando la memoria de nuestro interprete o nuestra PC se acabase.
Este ejemplo nos sirve para entender que para poder utilizar una función recursiva, necesitamos si o si un punto, en el que el bucle acabe, y deje de llamarse a si misma, tanto como una parte del codigo que nos acerque a ese punto de finalizacion.
Basicamente, una funcion recursiva tiene que tener tres partes, la primera es el caso base, que es la condicion terminal, la que define el fin del bucle.
La segunda es el paso que nos acerca a nuestro caso base y la tercera es el paso recursivo, en el que la funcion se invoca a si misma con una entrada reducida.

Si nos fijamos, la recursion es como la iteracion, asi que, cualquier funcion que puedas definir recursivamente, puede definirse también, usando ciclos, dependiendo del caso, nosotros deberemos decidir cual es la más indicada, la mas eficiente, y la que nos ayuda a tener un codigo mas limpio.

Ahora vamos a ver diferentes ejemplos, y diferentes aplicaciones para este tipo de funciones, para intentar entender mejor el como funcionan.

Recursion con numeros:

Todas las funciones recursivas necesitan un caso base para poder terminar. Sin embargo, el simple hecho de añadir un caso base a nuestra funcion no evitara que esta se ejecute infinitamente. La funcion debe tener un paso para acercarnos al caso base. Por ultimo esta el paso recursivo. En el paso recursivo, el problema se reduce a una version mas pequeña del problema.

Supongamos que tenemos una funcion que sumara los numeros del 1 a un numero cualquiera. Por ejemplo, si quisieramos sumar del numero 1 al numero 4, la funcion sumaría 1+2+3+4.
Vamos a utilizar recursividad para hacerlo, como dijimos antes, primero tenemos que determinar el caso base, en este ejemplo es cuando nuestro numero "n" es igual a cero, ya que en ese punto, ya habría sumado todos los valores entre el 4 y el 1.

En cada paso restaremos uno al numero actual, lo que acercará la función al caso base y el caso

recursivo será la funcion suma invocada con el numero reducido.

```
function sum(num){
  if (num === 0) {
    return 0;
  } else {
    return num + sum(--num)
  }
}
```

Recursion con arrays:

Las recursiones con arrays son similares a las con numeros, pero en vez de reducir el numero en cada paso, vamos a reducir el array hasta que lo hayamos vaciado.
Piensen una funcion suma, que recibe un array de numeros como entrada y devuelve la suma de todos los elementos de dicho array.
En este caso, nuestro caso base sería cuando el array esta vacio, o sea, array.length === 0
Lo que nos acerca al caso base es el arr.shift(), que por cada vez que ejecutamos la funcion, quita un valor del array
Y por ultimo el caso recursivo será cuando llamamos a la funcion otra vez con el array reducido sum(arr)

```
function sumaValores(array){
    if (arr.length === 0) {
      return 0;
    } else {
      return array.shift() + sumaValores(array);
    }
  }
```

Entonces, repasando, una funcion recursiva tiene tres partes, la primera es el caso base, que es la condicion terminal, la segunda el paso que nos acerca a nuestro caso base y la tercera es el paso recursivo, en el que la funcion se invoca a si misma con una entrada reducida.
Cualquier funcion recursiva, puede reescribirse con una estructura ciclica cualquiera.