



FULL STACK

**Comenzamos en unos
minutos**

ACADEMY
by NUMEN

Javascript: Clase 5

Programación Funcional

La programación funcional es un estilo de programación donde las soluciones son funciones simples y aisladas, sin efectos secundarios fuera del alcance de la función:
INPUT -> PROCESS -> OUTPUT

La programación funcional se trata de:

1. Funciones aisladas: no depende del estado del programa, que incluye variables globales que están sujetas a cambios.
1. Funciones puras: la misma entrada siempre da la misma salida
1. Funciones con efectos secundarios limitados: cualquier cambio o mutación en el estado del programa fuera de la función se controla cuidadosamente

Callbacks

Callbacks

Una función de tipo **Callback** nos permite pasar otra función como argumento e invocarla a través de esta cuantas veces se requiera.

En esta ocasión podemos observar como las funciones **saludarUsuario()** y **despedirUsuario()** se pasan como parámetro en la función **crearSaludo()** y luego se invocan al llamar a esta última.

```
function saludarUsuario(usuario) {  
    return `Hola ${usuario}!`  
}  
  
function despedirUsuario(usuario) {  
    return `Adiós ${usuario}!`  
}  
  
function crearSaludo(usuario, callback) {  
    return callback(usuario)  
}  
  
crearSaludo('Numen', saludarUsuario);  
crearSaludo('Numen', despedirUsuario);
```



```
firstTask(data, function(err, result) {  
  secondTask(data, function(err, result) {  
    thirdTask(data, function(err, result) {  
      fourthTask(data, function(err, result) {  
        fifthTask(data, function(err, result) {  
          // Code  
        });  
      });  
    });  
  });  
});
```

**CALLBACK
HELL**

Métodos Callback

.forEach()

forEach es un método que nos permite iterar arreglos, elemento por elemento, a través de una ejecución repetitiva de funciones.

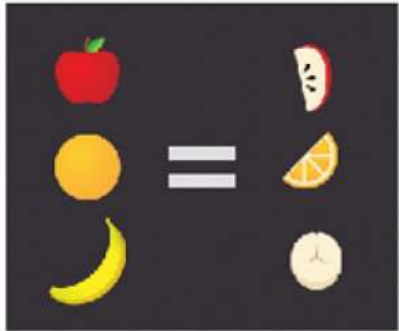
```
const profesores = ['Cinthia', 'Matias', 'Andres',  
  'Santi']  
  
// sin callback  
  
for (let i = 0; i < profesores.length; i++) {  
  console.log(profesores[i])  
}
```

```
const profesores = ['Cinthia', 'Matias', 'Aaron',  
  'Santi']  
  
// con callback  
  
profesores.forEach(function(elemento, indice) {  
  console.log(elemento)  
})
```

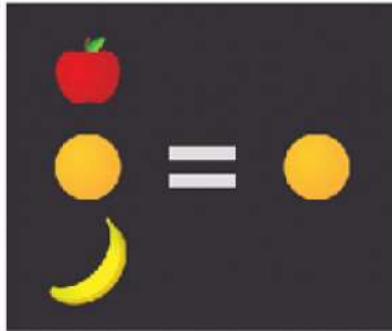
Aquí se puede observar como se ha reemplazado la iteración, a través de un ciclo for, de un arreglo por una función forEach.

Otras funciones callback...

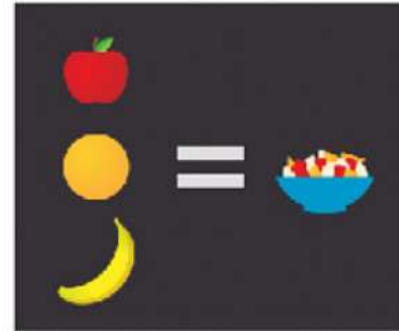
`Array.map()`



`Array.filter()`



`Array.reduce()`



.map()

El método `map()` nos permite iterar los elementos de un arreglo, transformarlos y devolver un nuevo arreglo con las transformaciones realizadas.

```
const profesores = [  
  {  
    nombre: 'Matias',  
    edad: '33',  
    profesion: 'Profesor',  
  },  
  {  
    nombre: 'Cinthia',  
    edad: '31',  
    profesion: 'Coordinadora',  
  },  
  {  
    nombre: 'Andres',  
    edad: '27',  
    profesion: 'Profesor',  
  },  
  {  
    nombre: 'Guillermo',  
    edad: '25',  
    profesion: 'Tutor',  
  }  
]
```

```
profesores.map(elemento => {  
  return (`  
    <h2>Bienvenidos a Academia  
    Numen</h2>  
  
    <p>En esta ocasión quiero  
    presentarles  
    a ${elemento.nombre} quien  
    será su ${elemento.profesion}  
    a lo largo de este curso.</p>  
  `)  
})
```

Diferencia entre forEach() y map()

```
const animales = ["Leon", "Paloma", "Tiburon", "Almeja", "Ganso"]  
  
// Ambas funciones recorren un arreglo  
  
// El forEach no devuelve un nuevo arreglo  
animales.forEach((animal, i) => animales[i] = `${animal}`)  
  
// El map devuelve un nuevo arreglo  
animales.map(animal => `${animal}`)
```

.reduce()

El método `reduce()` nos permite recorrer un arreglo y nos devuelve un único elemento.

Ejemplo: Sumar todos los elementos (numéricos) de un arreglo y devolvernos el resultado de esa suma.

```
const numeros = [1, 2, 3, 4, 5, 6, 7];

// sin callbacks

let suma = 0;
for (let i = 0; i < numeros.length; i++) {
    suma = suma + numeros[i];
}

// callbacks

const sumaReduce = numeros.reduce(function(accumulator,
elemento) {
    return accumulator + elemento;
}, 0);
```

.filter()

Como indica su nombre, el método **filter()** se utiliza para filtrar contenido de un arreglo, es decir, extraer solo ciertos elementos que coincidan con la condición especificada.

```
var palabras = ['chancleta', 'pato', 'bigote', 'ornitorrinco', 'termo', 'ajedrez'];  
  
var resultado = palabras.filter(palabra => palabra.length > 6);  
  
console.log(resultado);
```

.find()

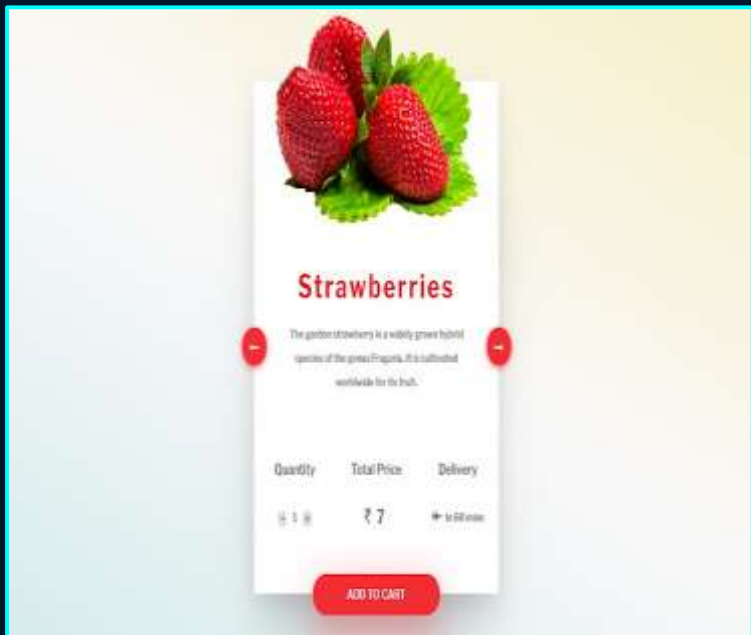
El método **.find()** es similar al **filter**, con la diferencia que en lugar de un arreglo, devuelve únicamente el primer elemento que coincide con la condición.

```
const frutas = ["Pera", "Banana", "Manzana", "Kiwi", "Mora"]
```

```
const encontrar = frutas.find(fruta => fruta.length === 4)
```

MAP: ejemplo práctico

En este ejemplo, mapearemos un arreglo de productos y en base a ello imprimiremos tarjetas de productos simuladas con los datos de cada objeto del arreglo.



```
const products = [  
  {id: 1, name: "Producto A", price: 10},  
  {id: 2, name: "Producto B", price: 20},  
  {id: 3, name: "Producto C", price: 30},  
  {id: 4, name: "Producto D", price: 40}  
]  
  
products.map(product => {  
  console.log(`  
    ${product.name}  
    ${product.price}  
  `)  
})
```

FIND: ejemplo práctico

En este ejemplo, usaremos el método `.find()` para buscar en el arreglo de productos del ejemplo anterior, un producto que coincida con el ID del botón al que le demos click.

```
// index.html
<button id="1" onclick="dispararId(id)">Producto A</button>
<button id="2" onclick="dispararId(id)">Producto B</button>
<button id="3" onclick="dispararId(id)">Producto C</button>
<button id="4" onclick="dispararId(id)">Producto D</button>
```

```
// index.js
function dispararId(id) {
  const newProduct = products.find(product => product.id == id)

  console.log(newProduct)
}
```

FILTER: ejemplo práctico



En este ejemplo, usaremos el método `.filter()` para eliminar del arreglo de productos al producto correspondiente al botón al que le demos click.

```
// index.html
<button id="1" onclick="dispararId(id)">Eliminar A</button>
<button id="2" onclick="dispararId(id)">Eliminar B</button>
<button id="3" onclick="dispararId(id)">Eliminar C</button>
<button id="4" onclick="dispararId(id)">Eliminar D</button>
```

```
// index.js
function dispararId(id) {
  const deleteProduct = products.filter(product => product.id !== id)

  console.log(deleteProduct)
}
```

Closures

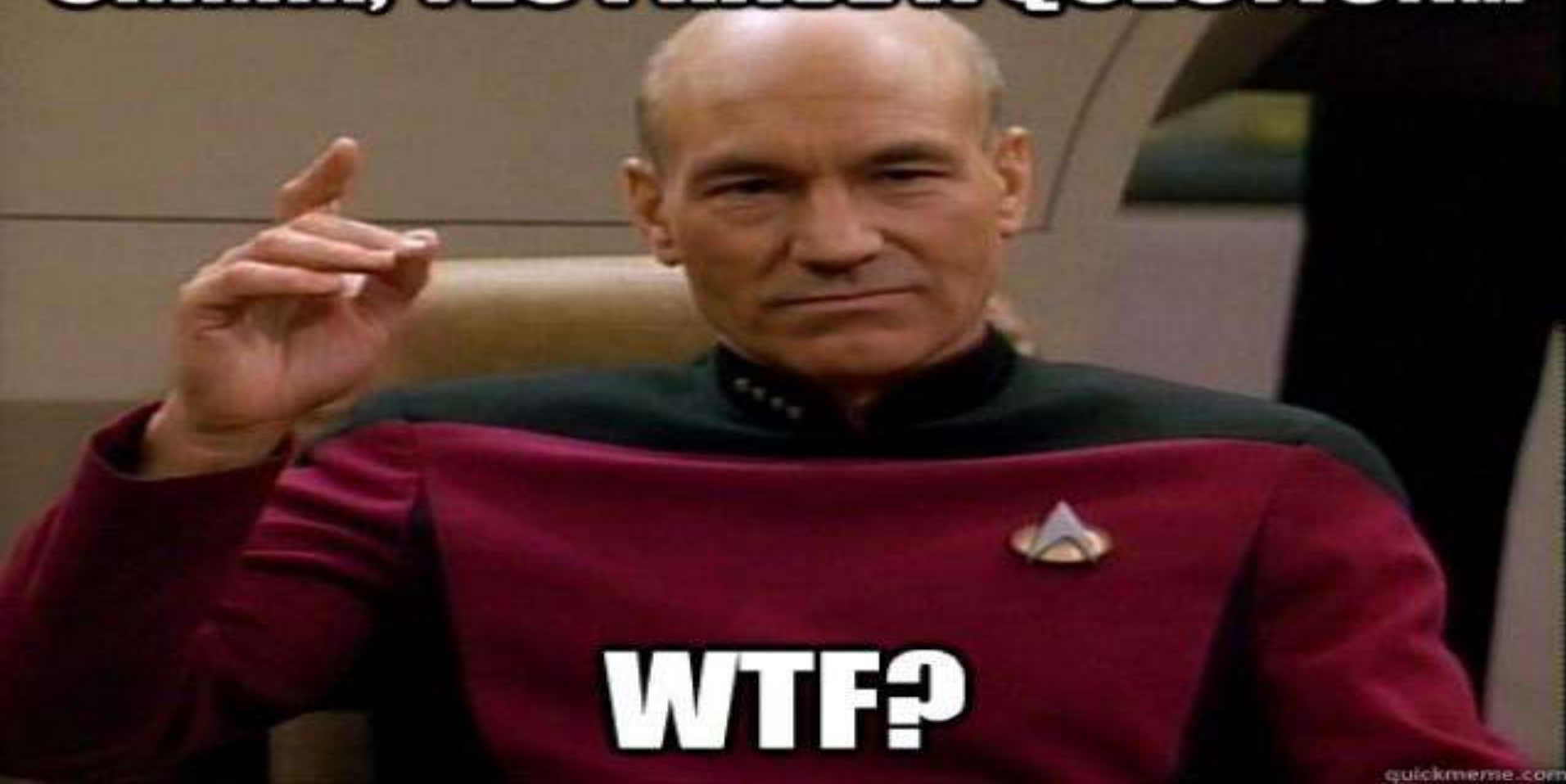
Closures

Las **closures** nos permiten **acceder** a variables que no estén exactamente definidas dentro del **scope** de una función.

Es decir, van a estar en otro contexto que ya no existirá más pero que Javascript nos reserva esa referencia para poder acceder a ella de todos modos.

```
function saludar( saludo ){  
    return function( nombre ){  
        console.log(` ${saludo} ${nombre} `);  
    }  
}  
  
var saludoHola = saludar('Hola'); // Esto devuelve  
una función  
  
saludoHola('Numen'); // 'Hola Numen'
```

UMMM, YES I HAVE A QUESTION...



WTF?

**NOT SURE IF MY
FUNCTION**

**OR A FUNCTION FROM INSIDE MY
FUNCTION**

Veamos paso a paso lo que va a ocurrir cuando ejecutemos este código. Primero se creará el **contexto de ejecución global**, en esta etapa el intérprete guardará espacio para la declaración de la función **saludar**. Luego, cuando se encuentra con la invocación a la función **saludar**, va a crear un nuevo contexto, y como vemos dentro de ese contexto la variable **saludo** va a tomar el valor que le pasamos por parámetro: **'Hola'**. El stack quedaría como está representado en la primera parte de la figura de abajo



Luego de terminar de ejecutar y retornar una función (la que estamos guardando en `saludar('Hola')`), ese contexto es destruido. Pero, ¿qué pasa con la variable `saludo`? El intérprete saca el contexto del stack, pero deja en algún lugar de memoria las variables que se usaron dentro (hay un proceso dentro de JavaScript que se llama garbage collection que eventualmente las va limpiando si no las usamos.). Por lo tanto, esa variable todavía va a estar en memoria (Segunda parte de la imagen). Por último ejecutamos la función `saludoHola`, y pasamos como parámetro el string `'Numen'`. Por lo tanto se crea un nuevo contexto de ejecución, con la variable mencionada. Ahora, cómo dentro de la función `saludoHola` hacemos referencia a la variable `saludo`, el intérprete intenta buscarla en su scope; Como `saludo` no está definida en ese contexto, el intérprete sale a buscarla siguiente la scope chain y a pesar de que el contexto ya no existe, la referencia al ambiente exterior y a sus variables todavía existe, a este fenómeno es que le llamamos **CLOSURE**. En el ejemplo, el closure está definido por el rectángulo punteado de rojo. Las closures no son algo que se escriban, o qué se le indique al intérprete, simplemente son una feature del lenguaje, simplemente ocurren. Nosotros no tenemos que pensar ni ocuparnos de mantener variables en memoria según el contexto de ejecución en el que estemos, el intérprete se encargará de esto siempre.



```
function saludar( saludo ){  
    return function( nombre ){  
        console.log(`${saludo} ${nombre}`);  
    }  
}  
  
var saludoHola = saludar('Hola'); // Esto devuelve una función  
  
saludoHola('Numen'); // 'Hola Numen'
```

Veamos paso a paso lo que va a ocurrir cuando ejecutemos este código. Primero se creará el **contexto de ejecución global**, en esta etapa el intérprete guardará espacio para la declaración de la función **saludar**.

Saludar Execution Context

saludo
'Hola'

Global Execution Context

saludo
'Hola'

Global Execution Context

SaludarHola Execution Context

nombre
'Numen'

saludo
'Hola'

Global Execution Context

```
function saludar( saludo ){  
  return function( nombre ){  
    console.log(`${saludo} ${nombre}`);  
  }  
}  
  
var saludoHola = saludar('Hola'); // Esto devuelve una función  
  
saludoHola('Numen'); // 'Hola Numen'
```

Luego, cuando se encuentra con la invocación a la función **saludar**, va a crear un nuevo contexto, y como vemos dentro de ese contexto la variable **saludo** va a tomar el valor que le pasamos por parámetro: **'Hola'**.

Saludar Execution Context

saludo
'Hola'

Global Execution Context

saludo
'Hola'

Global Execution Context

SaludarHola Execution Context

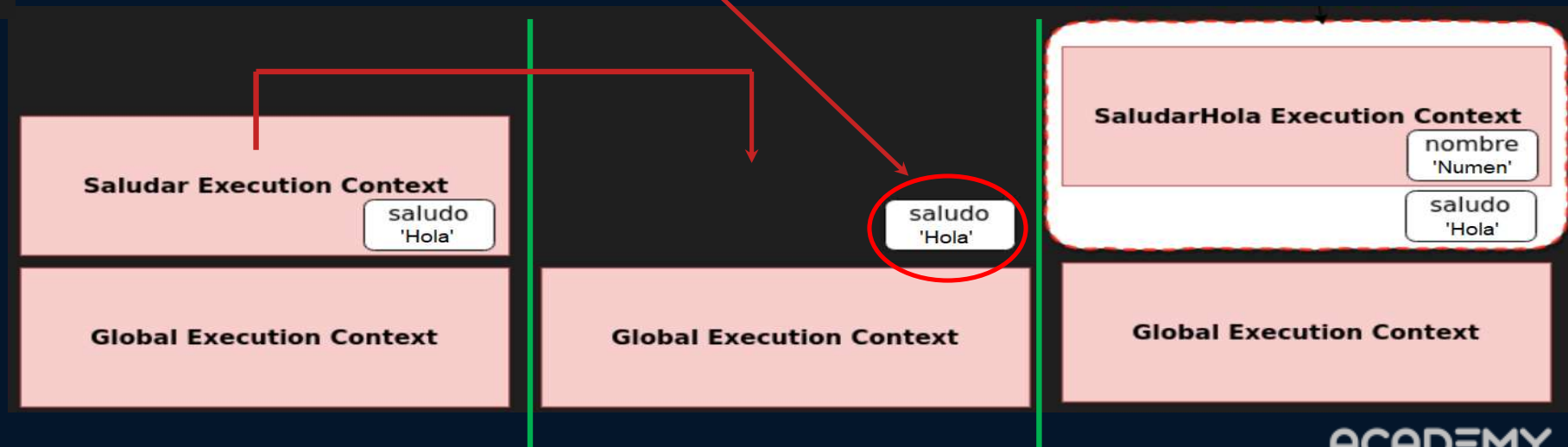
nombre
'Numen'

saludo
'Hola'

Global Execution Context

```
function saludar( saludo ){  
  return function( nombre ){  
    console.log(`${saludo} ${nombre}`);  
  }  
}  
  
var saludoHola = saludar('Hola'); // Esto devuelve una función  
  
saludoHola('Numen'); // 'Hola Numen'
```

Luego de terminar de ejecutar y retornar una función (la que estamos guardando en `saludar('Hola')`), ese contexto es destruido. ¿Pero qué pasa con la variable `saludo`? El intérprete saca el contexto del stack, pero deja en algún lugar de memoria las variables que se usaron dentro (hay un proceso dentro de JavaScript que se llama *garbage collection* que eventualmente las va limpiando si no las usamos). Por lo tanto, esa variable todavía va a estar en memoria (Segunda parte de la imagen).



```
function saludar( saludo ){
  return function( nombre ){
    console.log(`${saludo} ${nombre}`);
  }
}

var saludoHola = saludar('Hola'); // Esto devuelve una función

saludoHola('Numen'); // 'Hola Numen'
```

Por último ejecutamos la función **saludoHola**, y pasamos como parámetro el string **'Numen'**. Por lo tanto se crea un nuevo contexto de ejecución, con la variable mencionada.




```
function saludar( saludo ){
  return function( nombre ){
    console.log(`${saludo} ${nombre}`);
  }
}

var saludoHola = saludar('Hola'); // Esto devuelve una función

saludoHola('Numen'); // 'Hola Numen'
```

Ahora, cómo dentro de la función `saludoHola` hacemos referencia a la variable `saludo`, el intérprete intenta buscarla en su scope; Como `saludo` no está definida en ese contexto, el intérprete sale a buscarla siguiendo la scope chain y a pesar de que el contexto ya no existe, la referencia al ambiente exterior y a sus variables todavía existe, a este fenómeno es que le llamamos **CLOSURE**. En el ejemplo, el closure está definido por el rectángulo punteado de rojo. Las closures no son algo que se escriban, o qué se le indique al intérprete, simplemente son una feature del lenguaje, simplemente ocurren. Nosotros no tenemos que pensar ni ocuparnos de mantener variables en memoria según el contexto de ejecución en el que estemos, el intérprete se encargará de esto siempre.



Closures ☒ Fábrica de funciones

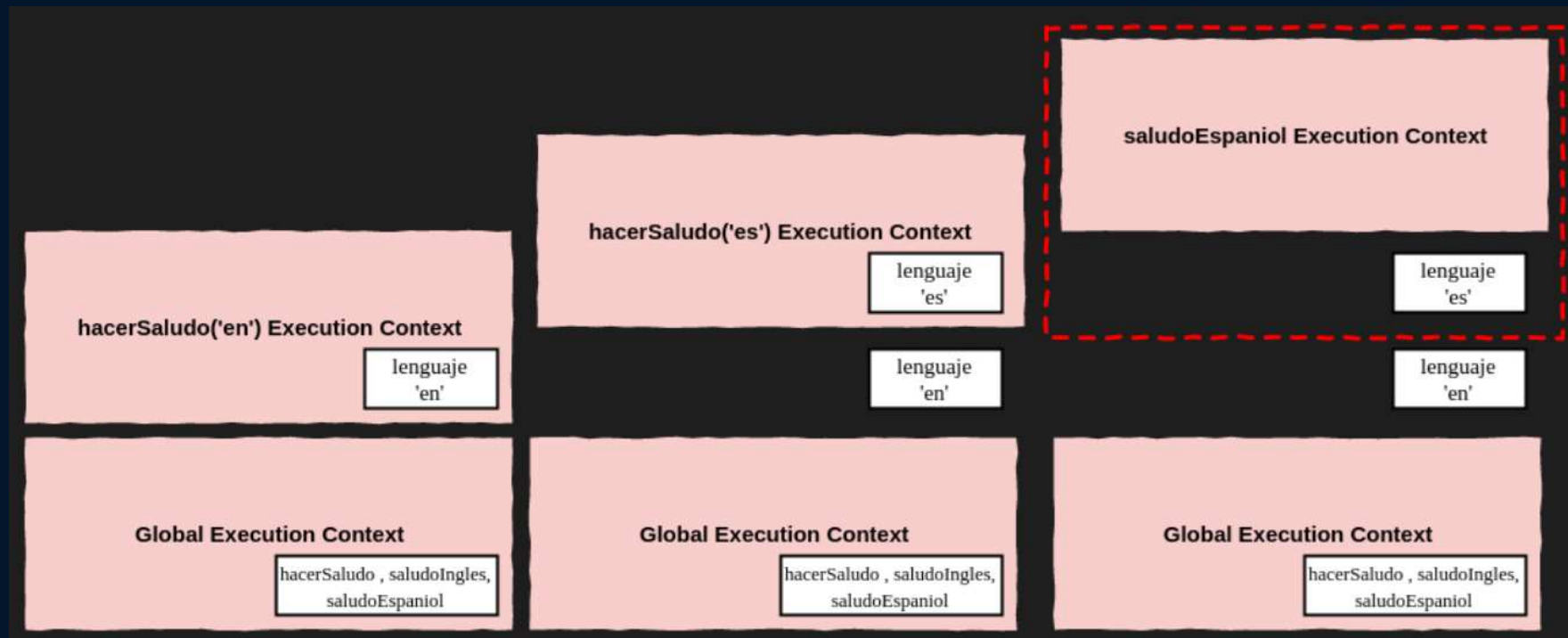
```
function hacerSaludo( lenguaje ) {  
  if ( lenguaje === 'en' ) {  
    return function () {  
      console.log('Hi!')  
    }  
  }  
  
  if (lenguaje === 'es') {  
    return function () {  
      console.log('Hola!')  
    }  
  }  
}  
  
let saludoIngles = hacerSaludo('en');  
let saludoEspañol = hacerSaludo('es');
```

Veamos el siguiente código, primero definimos una función que va retornar otra función (esta sería nuestra *fábrica de funciones*), esta recibe como parámetro el lenguaje del saludo, y retorna una función que salude (regresa un `console.log`) en el idioma recibido.

Si pensamos que ocurre cuando ejecutamos estas líneas, vamos a ver que se crearon dos closures. Uno para cada ejecución de la función `hacerSaludo`, en un closure la variable `lenguaje` contiene `'es'` y en el otro contiene `'en'`. Entonces, cuando invoquemos las funciones `saludoIngles` o `saludoEspañol`, el intérprete va a salir a buscar la referencia a esa variable fuera del contexto de ejecución y la va a encontrar en el closure correspondiente.

O sea, que estamos usando el concepto de closure para setear un parámetro para que viva sólo dentro de una función, además nadie puede ingresar al valor de lenguaje, esto agrega un poco de seguridad a nuestro código.

Cada vez que invocamos una función se genera un **execution context** para esa ejecución.
Si invocamos muchas veces la misma función ocurre lo mismo.



El dilema del contador

En este ejemplo, construiremos un contador común y corriente, el cual conectaremos con un botón a modo de que, cada vez que demos click en el mismo, imprimamos en la consola una variable cuyo valor irá incrementándose con cada click.



```
// index.js  
let counter = 0;  
  
const add = () => {  
    return counter += 1  
}  
  
const handleCounter = () => {  
    console.log(add())  
}
```

```
// index.html  
<button onclick="handleCounter()">Count!</button>
```

Pero nuestro contador tiene un problema. Al tener la variable **counter** en el **ámbito global**, esta podría ser modificada desde cualquier parte de nuestro programa sin necesidad de ejecutar la función **add()**. Este es un comportamiento que no deseamos.

```
// index.js
const add = () => {
  let counter = 0;
  return counter += 1
}

const handleCounter = () => {
  console.log(add())
}
```

Si intentamos colocar la variable dentro de la función notaremos que esta se rompe, no permitiendo incrementar más de 1.

Esto sucede porque, con cada invocación, nuestra variable actualizada se resetea.

Para poder evitar que se resetee necesitaremos usar las **CLOSURES!!**

```
// index.js
const add = () => {
  let counter = 0;
  return function () {
    counter += 1
    return counter
  }
}

const increase = add()

const handleCounter = () => {
  console.log(increase())
}
```

Para evitar que se resetee nuestra variable actualizada, debemos encerrarla en un ámbito tan privado que, al ejecutar la función **add()** no se pueda tener acceso a ella de manera directa.

Para ello creamos una clausura dentro de **add()** donde pueda descansar nuestra variable.

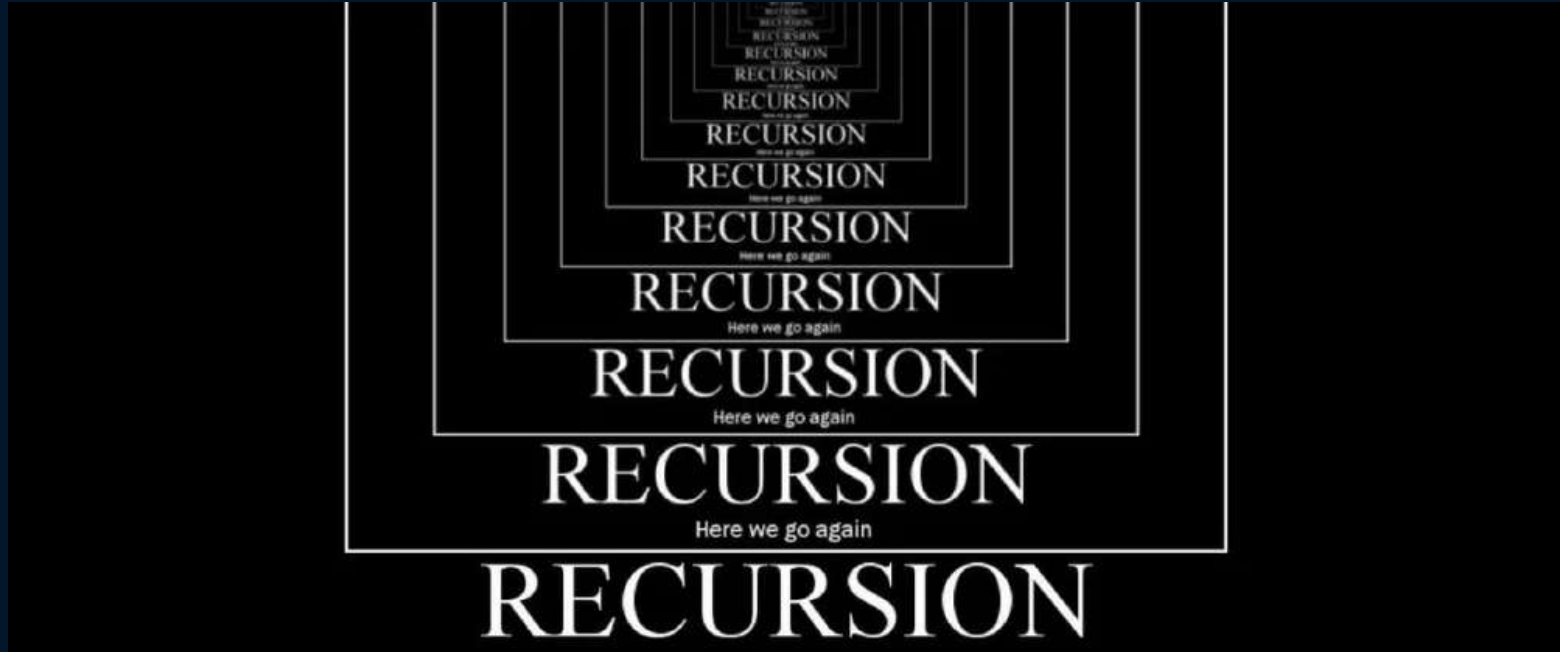
De este modo, solo accederemos a ella para actualizar el valor pero esta volverá a 0 con cada llamado.



MASTERING CLOSURE AND HOISTING

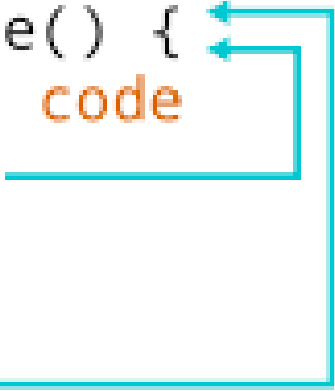
Recursividad

Un algoritmo recursivo es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí misma se conoce como llamada recursiva o recurrente.



Un función recursiva se vería así:

```
function recurse() {  
    // function code  
    recurse();  
}  
  
recurse();
```



function
call

Veamos un ejemplo concreto más en detalle

Contador recursivo

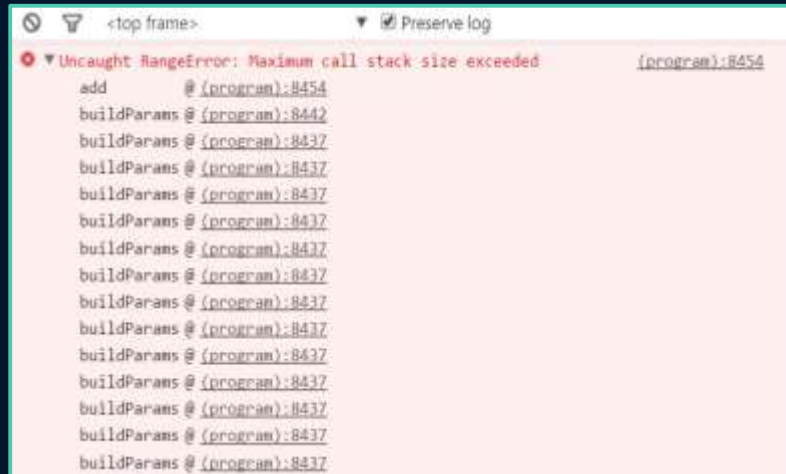
La idea de este ejercicio es programar una cuenta regresiva usando recursividad. Como podemos observar esta función únicamente imprime el argumento que ingresa por parámetro.

```
const cuentaRegresiva = numero => {  
  console.log(numero)  
}  
  
cuentaRegresiva(3)
```



Si intentamos realizar la cuenta regresiva por medio de una **recursión**, notaremos no sólo que se genera un loop, sino que además este terminará en un **error**.

```
const cuentaRegresiva = numero => {  
  console.log(numero)  
  
  cuentaRegresiva(numero - 1)  
}  
  
cuentaRegresiva(3)
```



Esto sucede porque no especificamos una **condición de corte**, algo que toda función recursiva necesita.

Para solucionar el problema anterior, simplemente definimos una variable que contenga nuestra operación de decremento y luego mediante una estructura de control impedimos que se siga ejecutando.

```
const cuentaRegresiva = numero => {  
  console.log(numero)  
  
  let proximoNumero = numero - 1;  
  
  if (proximoNumero > 0) {  
    cuentaRegresiva(proximoNumero);  
  }  
}  
  
cuentaRegresiva(3)
```

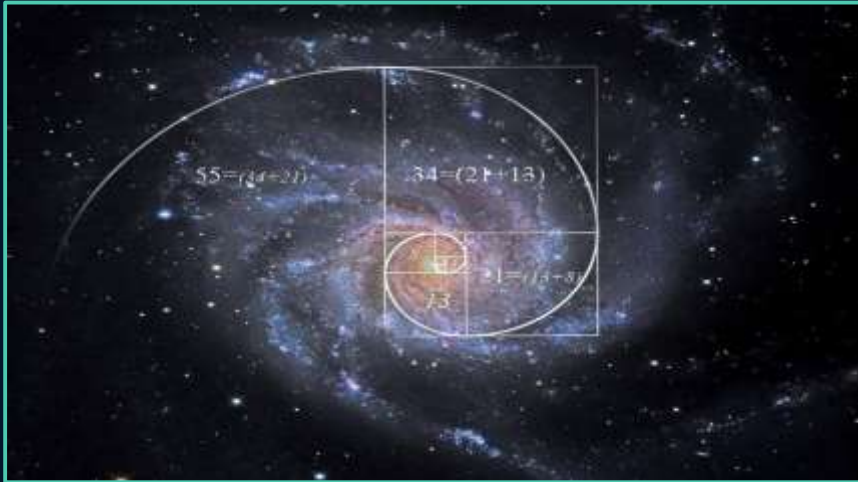
Este tipo de lógica es ideal para controlar cualquier clase de funcionalidad que implique una cuenta regresiva, como por ejemplo, la detonación de una bomba o el clásico contador que indica la llegada del año nuevo.



Secuencia de Fibonacci

Se trata de una secuencia infinita de números naturales; a partir del 0 y el 1, se van sumando a pares, de manera que cada número es igual a la suma de sus dos anteriores, de manera que:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...



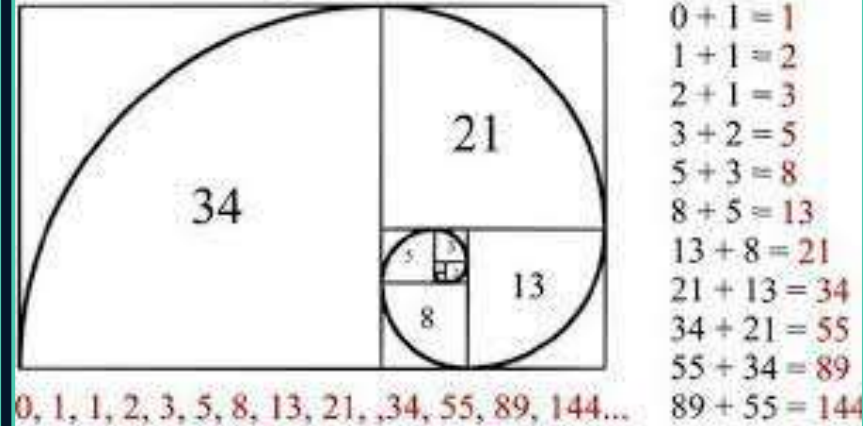
En este ejercicio intentaremos programar esta secuencia usando recursividad. Si bien nos llevará unas pocas líneas de código, esta será complicada de comprender.

Prestar mucha atención!!!!

Aquí podemos apreciar la fórmula matemática detrás de la famosa secuencia y como va incrementándose a medida que cambia el valor de N.

Es importante excluir 0 y 1 ya que no hay 2 valores de referencia previos para ellos.

```
// Formula:  $F_n = F_{(n-2)} + F_{(n-1)}$  donde  $n \geq 2$ .  
// 0, 1, 1, 2, 3, 5, 8...  
  
const fibonacci = n => {  
    if(n < 2) return n;  
  
    return fibonacci(n - 2) + fibonacci(n - 1)  
}
```



Por medio de la recursividad, logras llamar a valores previos en la secuencia para poder calcular los siguientes valores.

ACADEMY
by NUMEN